TURING 图灵程序设计丛书

Packt

Python High Performance
Second Edition

# Python高性能

（第2版）

[加] 加布丽埃勒·拉纳诺 著　袁国忠 译

利用并发和分布式处理技术构建高性能、可伸缩的Python应用程序

中国工信出版集团　人民邮电出版社
POSTS & TELECOM PRESS

**加布丽埃勒·拉纳诺**

（Gabriele Lanaro）

数据科学家、软件工程师，对机器学习、信息检索、数值计算可视化、Web开发、计算机图形学和系统管理有浓厚的兴趣。开源软件包chemlab和chemview的开发者。现就职于Tableau软件公司。

# 数字版权声明

TURING 图灵程序设计丛书

Python High Performance
Second Edition

# Python高性能

（第2版）

[加] 加布丽埃勒·拉纳诺 著
袁国忠 译

人民邮电出版社
北京

图书在版编目（C I P）数据

Python高性能 : 第2版 / (加) 加布丽埃勒·拉纳诺
(Gabriele Lanaro) 著 ; 袁国忠译. -- 北京 : 人民邮
电出版社, 2018.8
（图灵程序设计丛书）
ISBN 978-7-115-48877-0

Ⅰ. ①P… Ⅱ. ①加… ②袁… Ⅲ. ①软件工具－程序
设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2018)第155441号

## 内 容 提 要

本书主要介绍如何让 Python 程序发挥强大性能，内容涵盖针对数值计算和科学代码的优化，以及用于提高 Web 服务和应用响应速度的策略。具体内容有：基准测试与剖析、纯粹的 Python 优化、基于 NumPy 和 Pandas 的快速数组操作、使用 Cython 获得 C 语言性能、编译器探索、实现并发性、并行处理、分布式处理、高性能设计等。

本书适合 Python 开发人员阅读。

◆ 著　　　　[加] 加布丽埃勒·拉纳诺
译　　　　袁国忠
责任编辑　岳新欣
责任印制　周昇亮
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷
◆ 开本：800×1000　1/16
印张：12.25
字数：280千字　　　　2018年 8 月第 1 版
印数：1 – 3 000册　　　　2018年 8 月北京第 1 次印刷
著作权合同登记号　图字：01-2017-8618号

定价：59.00元

**读者服务热线：(010)51095186转600　印装质量热线：(010)81055316**

**反盗版热线：(010)81055315**

**广告经营许可证：京东工商广登字 20170147 号**

# 版权声明

# 前　　言

最近几年，Python 编程语言的人气急剧上升，其直观而有趣的语法及大量质量上乘的第三方库居功至伟。很多大学的编程入门和进阶课程，以及科学和工程等数值密集型领域，都选择将 Python 作为编程语言，它还被用于编写机器学习应用程序、系统脚本和 Web 应用程序。

大家普遍认为，Python 解释器参考版 CPython 比 C、C++和 Fortran 等低级语言效率低下。CPython 之所以性能糟糕，是因为程序指令没有编译成高效的机器码，而是由解释器处理。虽然使用解释器有些优点，如可移植性以及可省略编译步骤，但在程序和机器之间增加了一个间接层，降低了执行效率。

多年来，已制定出很多克服 CPython 性能缺点的策略。本书旨在填补这方面的空白，介绍如何让 Python 程序的性能始终强劲。

本书介绍如何优化数值计算和科学代码，还涵盖了缩短 Web 服务和应用程序响应时间的策略，这些对很多读者都极具吸引力。

本书可按顺序从头到尾地阅读，但其中的每章也自成一体，所以如果你已熟悉前面的主题，可直接跳到感兴趣的部分。

## 涵盖的内容

第 1 章介绍如何评估 Python 程序的性能，以及找出并隔离速度缓慢代码的实用策略。

第 2 章讨论如何使用 Python 标准库和第三方 Python 模块提供的高效数据结构和算法来缩短程序的执行时间。

第 3 章提供了 NumPy 和 Pandas 包的使用指南。掌握这些包后，你就可使用简洁而富有表达力的接口来实现快速的数值算法。

第 4 章是一个 Cython 教程，这种语言使用与 Python 兼容的语法来生成高效的 C 语言代码。

第 5 章介绍可用来将 Python 代码编译成高效机器码的工具。在该章中，你将学习如何使用

Numba 和 PyPy，其中前者是一个 Python 函数优化编译器，而后者是一个能够动态地执行并优化 Python 程序的解释器。

第 6 章提供了异步编程和响应式编程指南。你将学习重要的术语和概念，以及如何使用框架 `asyncio` 和 RxPy 编写整洁的并发代码。

第 7 章简要地介绍多核处理器和 GPU 并行编程。在该章中，你将学习如何使用模块 `multiprocessing` 以及 Theano 和 Tensorflow 来实现并行性。

第 8 章是前一章内容的延伸，专注于在分布式系统上运行并行算法来解决大型问题和大数据处理问题。该章还介绍了 Dask、PySpark 和 `mpi4py` 库。

第 9 章讨论通用的优化策略，以及开发、测试和部署高性能 Python 应用程序的最佳实践。

## 需要什么

本书的示例代码都在 Ubuntu 16.04 系统中使用 Python 3.5 进行了测试，但这些示例大都能够在 Windows 和 Mac OS X 操作系统上运行。

推荐使用 Anaconda 发行包来安装 Python 和相关的库，这个发行包有用于 Linux、Windows 和 Mac OS X 的版本，可从 https://www.continuum.io/downloads 下载。

## 为谁而写

本书适合想要改善应用程序的性能并掌握了 Python 基本知识的 Python 程序员阅读。

## 排版约定

为将不同类型的信息区分开来，本书使用了很多文本样式。下面列出其中一些样式及其含义。

正文中的代码、数据库表名、用户输入，使用如下样式：“总之，我们将实现一个名为 `ParticleSimulator.evolve_numpy` 的方法，并使用基准测试将其同纯粹的 Python 版本（更名为 `ParticleSimulator.evolve_python`）进行比较。”

代码块使用如下样式：

```
def square(x):
return x * x

inputs = [0, 1, 2, 3, 4]
outputs = pool.map(square, inputs)
```

要让你注意代码块的特定部分时，相关的代码行用粗体表示：

```
def square(x):
return x * x

inputs = [0, 1, 2, 3, 4]
outputs = pool.map(square, inputs)
```

命令行输入或输出使用如下样式：

```
$ time python -c 'import pi; pi.pi_serial()'
real 0m0.734s
user 0m0.731s
sys 0m0.004s
```

新术语和重要词语使用黑体字。



此图标表示警告或重要的注意事项。




此图标表示提示和技巧。

## 读者反馈

欢迎提供反馈，请将你对本书的看法告诉我们：哪些方面是你喜欢的，哪些方面你不喜欢。读者的反馈对我们来说很重要，因为这可帮助我们推出可最大限度发挥其功效的著作。

要给我们提供反馈，只需向 feedback@packtpub.com 发送电子邮件，并在主题中注明书名。

如果你有擅长的主题，并有志于写书或撰稿，请参阅 www.packtpub.com/authors 的撰稿指南。

## 客户支持

购买英文版图书后，你将获得各种帮助，让你购买的图书最大限度地发挥其功效。

## 下载示例代码

你可访问 http://www.packtpub.com 并使用你的账户下载本书的代码示例文件。如果你是在其他地方购买的本书，可访问 http://www.packtpub.com/support 并注册，以便我们将文件通过电子邮件发送给你。

要下载代码文件，可采取如下步骤。

(1) 访问我们的网站，使用电子邮件地址和密码注册并登录。
(2) 将鼠标指向页面顶部的标签 SUPPORT。
(3) 单击 Code Downloads & Errata。
(4) 在搜索框中输入书名。
(5) 选择要下载哪本书的代码文件。
(6) 从下拉列表中选择该书是在哪里购买的。
(7) 单击 Code Download。

下载文件后，使用下列软件的最新版解压缩：

- WinRAR / 7-Zip（Windows）；
- Zipeg / iZip / UnRarX（Mac）；
- 7-Zip / PeaZip（Linux）。

本书的示例代码还托管在 GitHub 上（https://github.com/PacktPublishing/Python-High-Performance-Second-Edition）。我们还在 https://github.com/PacktPublishing/提供了众多图书的示例代码以及视频，敬请访问！

## 勘误

我们万分小心，力图让图书的内容准确无误，即便如此，错误也在所难免。如果你在出版的图书中发现错误（无论是正文还是代码中的错误），请告诉我们，我们将感激不尽。这样做将让其他读者免遭同样的挫折，还可帮助我们改进该书的后续版本。无论你发现什么错误，都请告诉我们。为此，你可访问 http://www.packtpub.com/submit-errata，输入书名，单击链接 Errata Submission Form，再输入你发现的错误的详情。[①]你提交的勘误得到确认后，将被上传到我们的网站或添加到既有的勘误列表中。

要查看已提交的勘误，请访问 https://www.packtpub.com/books/content/support，并在搜索框中输入书名，Errata 栏将列出你搜索的信息。

## 打击盗版

在网上发布盗版材料是个屡禁不绝的问题。在保护版权和许可方面，本社的态度非常严肃，如果你在网上看到本社作品的非法复制品，请马上把网址或网站名告诉我们，以便我们采取补救措施。

① 中文版可访问图灵社区本书主页 www.ituring.com.cn/book/2006 提交勘误。——编者注

请通过 copyright@packtpub.com 与我们取得联系，并提供你怀疑的盗版材料的链接。

对于你为保护我们的作者和提供有价值内容的能力提供的帮助，我们感激不尽。

## 问题

无论你有什么与本书相关的问题，都可通过 questions@packtpub.com 与我们联系，我们将竭尽全力去解决。

## 电子书

扫描如下二维码，即可购买本书电子版。

# 致　　谢

感谢Packt出版社Vikas Tiwari等编辑的支持；感谢我的女朋友Harani忍受我长时间挑灯写作；感谢朋友们自始至终的陪伴和支持；还要感谢父母给我机会追求自己的理想。

最后，感谢百怡咖啡赋予我写作本书的动力。

# 目　　录

第 1 章

# 基准测试与剖析

就提高代码速度而言，最重要的是找出程序中速度缓慢的部分。所幸在大多数情况下，导致应用程序速度缓慢的代码都只占程序的很小一部分。确定这些关键部分后，就可专注于需要改进的部分，避免将时间浪费于微优化。

通过**剖析**（profiling），可确定应用程序的哪些部分消耗的资源最多。**剖析器**（profiler）是这样一种程序：运行应用程序并监控各个函数的执行时间，以确定应用程序中哪些函数占用的时间最多。

Python 提供了多个工具，可帮助找出瓶颈并度量重要的性能指标。本章将介绍如何使用标准模块 `cProfile` 和第三方包 `line_profiler`，还将介绍如何使用工具 `memory_profiler` 剖析应用程序的内存占用情况。本章还将介绍另一个很有用的工具——KCachegrind，使用它能以图形化方式显示各种剖析器生成的数据。

**基准测试程序**（benchmark）是用于评估应用程序总体执行时间的小型脚本。本章将介绍如何编写基准测试程序以及如何准确地测量程序的执行时间。

本章介绍如下主题：

- 通用的高性能编程原则；
- 编写测试和基准测试程序；
- Unix 命令 `time`；
- Python 模块 `timeit`；
- 使用 `pytest` 进行测试和基准测试；
- 剖析应用程序；
- 标准工具 `cProfile`；
- 使用 KCachegrind 解读剖析结果；
- 工具 `line_profiler` 和 `memory_profiler`；
- 使用模块 `dis` 对 Python 代码进行反汇编。

## 1.1　设计应用程序

就设计高性能程序而言，最重要的是在编写代码期间不进行细微的优化。

> “过早优化是万恶之源。”
>
> ——高德纳

在开发过程的早期阶段，程序的设计可能瞬息万变，你可能需要大规模地改写和重新组织代码。在此阶段，你需要对不同的原型进行测试，而不进行优化，这样可自由地分配时间和精力，确保程序能够得到正确的结果，同时具有灵活的设计。归根结底，谁都不想要一个运行速度很快但结果却不正确的应用程序。

优化代码时，必须牢记如下箴言。

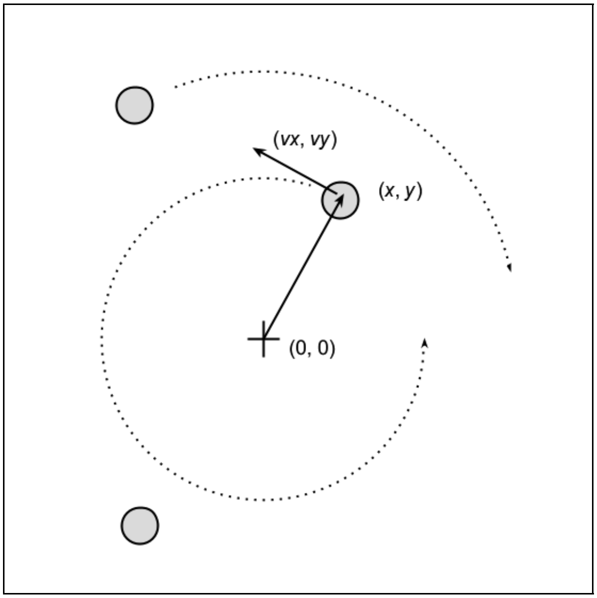
- **让它能够运行**：必须让软件能够运行，并确保它生成的结果是正确的。这个探索阶段让你能够对应用程序有更深入的认识，并在早期发现重大设计问题。
- **确保设计正确**：必须确保程序的设计是可靠的。进行任何性能优化前务必先重构，这可帮助你将应用程序划分成独立而内聚且易于维护的单元。
- **提高运行速度**：确保程序能够运行且结构优良后，就可专注于性能优化了。例如，如果内存消耗是个问题，你可能想对此进行优化。

在本节中，我们将编写一个**粒子模拟器**测试应用程序并对其进行剖析。这个**模拟器**程序接受一些粒子，并根据我们指定的规则模拟这些粒子随时间流逝的运动情况。这些粒子可能是抽象实体，也可能是真实的物体，如运动的桌球、气体中的分子、在太空中移动的星球、烟雾颗粒、液体等。

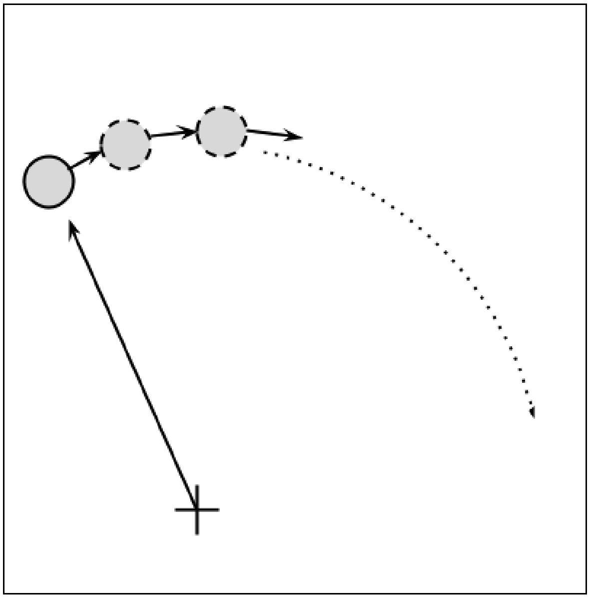
在物理、化学、天文学等众多学科中，计算机模拟都很有用。对用于模拟系统的应用程序来说，性能非常重要，因此科学家和工程师会花费大量时间来优化其代码。为了研究真实的系统，通常必须模拟大量的实体，因此即便是细微的性能提升也价值不菲。

在这个模拟系统示例中，包含的粒子以不同的速度绕中心点不断地旋转，就像钟表的指针一样。

为了模拟这种系统，需要如下信息：粒子的起始位置、速度和旋转方向。我们必须根据这些信息计算粒子在下一个时刻的位置。下图说明了这个系统，其中原点为(0, 0)，位置用向量 *x* 和 *y* 表示，而速度用向量 *vx* 和 *vy* 表示。




圆周运动的基本特征是，粒子的运动方向始终与其当前位置到中心点的线段垂直。要移动粒子，只需采取一系列非常小的步骤（对应于系统在很短时间内的变化），并在每个步骤中都根据粒子的运动方向修改其位置，如下图所示。



我们将以面向对象的方式设计这个应用程序。根据这个应用程序的需求，显然需要设计一个

通用的 Particle 类，用于存储粒子的位置（x 和 y）以及角速度（ang_vel）。

```
class Particle:
    def __init__(self, x, y, ang_vel):
        self.x = x
        self.y = y
        self.ang_vel = ang_vel
```

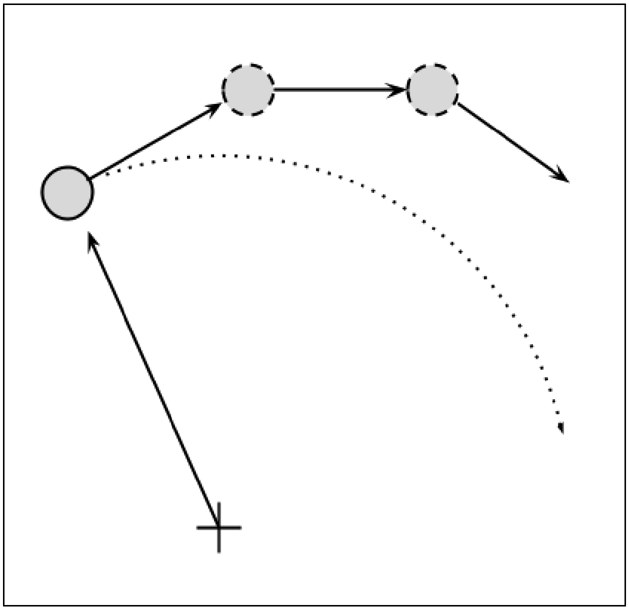
请注意，这里所有的参数都可正可负，其中 ang_vel 的符号决定了旋转方向。

还需要设计另一个类——ParticleSimulator，它封装了运动定律，负责随时间流逝修改粒子的位置。在这个类中，方法__init__存储一个 Particle 实例列表，而方法 evolve 根据指定的定律修改粒子的位置。

我们要让粒子绕坐标(0, 0)以固定的速度旋转，而运动方向总是与从粒子当前位置到中心点的线段垂直（参见本章的第一个图示）。要将运动方向表示为 x 和 y 向量（Python 变量 v_x 和 v_y），使用下面的公式即可。

```
v_x = -y / (x**2 + y**2)**0.5
v_y = x / (x**2 + y**2)**0.5
```

对于特定的粒子，经过时间 $t$ 后，它将到达圆周上的下一个位置。我们可以这样近似计算圆周轨迹：将时段 $t$ 分成一系列很小的时段 d$t$，在这些很小的时段内，粒子沿圆周的切线移动。这样就近似地模拟了圆周运动。为避免误差过大（如下图所示），时段 d$t$ 必须非常短。



简而言之，为计算经过时间 *t* 后的粒子位置，必须采取如下步骤。

(1) 计算运动方向（`v_x` 和 `v_y`）。

(2) 计算位移（`d_x` 和 `d_y`），即时段 d*t*、角速度和移动方向的乘积。

(3) 不断重复第(1)步和第(2)步，直到时间过去 *t*。

`ParticleSimulator` 类的完整实现代码如下：

```
class ParticleSimulator:

    def __init__(self, particles):
        self.particles = particles

    def evolve(self, dt):
        timestep = 0.00001
        nsteps = int(dt/timestep)

        for i in range(nsteps):
            for p in self.particles:
                # 1. 计算方向
                norm = (p.x**2 + p.y**2)**0.5
                v_x = -p.y/norm
                v_y = p.x/norm

                # 2. 计算位移
                d_x = timestep * p.ang_vel * v_x
                d_y = timestep * p.ang_vel * v_y

                p.x += d_x
                p.y += d_y
                # 3. 不断重复，直到时间过去 t
```

为了可视化这里的粒子，可使用 `matplotlib` 库。这个库不包含在 Python 标准库中，但可使用命令 `pip install matplotlib` 轻松地安装它。

> 也可使用发行包 Anaconda Python，它包含 `matplotlib` 以及本书使用的其他大多数第三方包。Anaconda 是免费的，可用于 Linux、Windows 和 Mac。

为了创建交互式可视化，我们使用函数 `matplotlib.pyplot.plot` 以点的方式显示粒子，并使用 `matplotlib.animation.FuncAnimation` 类以动画方式显示粒子随时间流逝的移动情况。

函数 `visualize` 将一个 `ParticleSimulator` 实例作为参数，并以动画方式显示粒子的运动轨迹。为了使用 `matplotlib` 来显示粒子的运动规则，必须采取的步骤如下。

- 创建并设置坐标轴，再使用函数 `plot` 来显示粒子。函数 `plot` 将 *x* 坐标和 *y* 坐标列表作为参数。

- 编写初始化函数 init 和函数 animate，其中后者使用方法 line.set_data 来更新 $x$ 和 $y$ 坐标。
- 创建一个 FuncAnimation 实例:传入函数 init 和 animate 以及参数 interval 和 blit，其中参数 interval 指定更新间隔，而 blit 可改善图像的更新速率。
- 使用 plt.show()运行动画。

```
from matplotlib import pyplot as plt
from matplotlib import animation

def visualize(simulator):

    X = [p.x for p in simulator.particles]
    Y = [p.y for p in simulator.particles]

    fig = plt.figure()
    ax = plt.subplot(111, aspect='equal')
    line, = ax.plot(X, Y, 'ro')

    # 指定坐标轴的取值范围
    plt.xlim(-1, 1)
    plt.ylim(-1, 1)

    # 这个方法将在动画开始时运行
    def init():
        line.set_data([], [])
        return line, # 这里的逗号必不可少!

    def animate(i):
        # 我们让粒子运动 0.01 个时间单位
        simulator.evolve(0.01)
        X = [p.x for p in simulator.particles]
        Y = [p.y for p in simulator.particles]

        line.set_data(X, Y)
        return line,

    # 每隔 10 毫秒调用一次动画函数
    anim = animation.FuncAnimation(fig,
                                   animate,
                                   init_func=init,
                                   blit=True,
                                   interval=10)
    plt.show()
```

为了测试这些代码，我们定义了一个简短的函数——test_visualize，它以动画方式模拟一个包含 3 个粒子的系统，其中每个粒子的运动方向各不相同。请注意，第三个粒子环绕一周的速度是其他两个粒子的 3 倍。

```
def test_visualize():
    particles = [Particle(0.3, 0.5, 1),
                 Particle(0.0, -0.5, -1),
```

```
                 Particle(-0.1, -0.4, 3)]

    simulator = ParticleSimulator(particles)
    visualize(simulator)

if __name__ == '__main__':
    test_visualize()
```

函数 `test_visualize` 很有用，可帮助你直观地理解系统随时间流逝的变化情况。在下一节，我们将再编写一些测试函数，以核实这个程序是正确的并测量其性能。

## 1.2 编写测试和基准测试程序

编写管用的模拟器后，便可着手测量其性能，并对代码进行优化，让模拟器能够处理尽可能多的粒子。首先，我们将编写测试和基准测试程序。

我们需要一个检查模拟结果是否正确的测试。为了优化程序，通常必须采取多种策略，但在反复重写代码的过程中，很容易引入 bug。可靠的测试集可确保每次迭代后实现都是正确的，这让我们能够大胆地进行不同的尝试，并深信只要能够通过测试集，代码就依然是像期望的那样工作的。

我们的测试将接受 3 个粒子，模拟 0.1 个时间单位，并将结果与来自参考实现的结果进行比较。为了组织测试，一种不错的方式是，对于应用程序的每个方面（或者说单元）都使用一个不同的函数进行测试。鉴于这个应用程序的功能都是在方法 `evolve` 中实现的，因此我们将把测试函数命名为 `test_evolve`。下面列出了函数 `test_evolve` 的实现代码。请注意，为了对浮点数进行比较，我们使用了函数 `fequal` 来确定它们的差在一定范围内。

```
def test_evolve():
    particles = [Particle( 0.3, 0.5, +1),
                 Particle( 0.0, -0.5, -1),
                 Particle(-0.1, -0.4, +3)]

    simulator = ParticleSimulator(particles)

    simulator.evolve(0.1)

    p0, p1, p2 = particles

    def fequal(a, b, eps=1e-5):
        return abs(a - b) < eps

    assert fequal(p0.x, 0.210269)
    assert fequal(p0.y, 0.543863)

    assert fequal(p1.x, -0.099334)
    assert fequal(p1.y, -0.490034)
```

```
        assert fequal(p2.x, 0.191358)
        assert fequal(p2.y, -0.365227)

if __name__ == '__main__':
    test_evolve()
```

测试可确保我们正确地实现了功能，但几乎没有提供任何有关运行时间的信息。基准测试程序是简单而有代表性的用例，可通过执行它来评估应用程序的运行时间。对于跟踪程序在每次迭代后运行速度有多快，基准测试程序很有用。

为了编写一个有代表性的基准测试程序，我们可实例化 1000 个坐标和角速度都是随机的 `Particle` 对象，并将它们提供给 `ParticleSimulator` 类，然后让系统运行 0.1 个时间单位。

```
from random import uniform

def benchmark():
    particles = [Particle(uniform(-1.0, 1.0),
                          uniform(-1.0, 1.0),
                          uniform(-1.0, 1.0))
                 for i in range(1000)]

    simulator = ParticleSimulator(particles)
    simulator.evolve(0.1)

if __name__ == '__main__':
    benchmark()
```

## 测量基准测试程序的运行时间

要计算基准测试程序的运行时间，一种非常简单的方法是使用 Unix 命令 `time`。通过像下面这样使用命令 `time`，可轻松地测量任何进程的执行时间。

```
    $ time python simul.py
real    0m1.051s
user    0m1.022s
sys     0m0.028s
```

在 Windows 系统中，没有命令 `time`。要在 Windows 系统中安装 Unix 工具，如命令 `time`，可使用 `cygwin` shell（可从其官网下载）。你也可使用类似的 PowerShell 命令来测量执行时间，如 `Measure-Command`。

默认情况下，`time` 显示 3 个指标。

- `real`：从头到尾运行进程实际花费的时间，与人用秒表测量得到的时间相当。
- `user`：在计算期间，所有 CPU 花费的总时间。
- `sys`：在执行与系统相关的任务（如内存分配）期间，所有 CPU 花费的总时间。

请注意，在有些情况下，`user` 与 `sys` 的和可能大于 `real`，这是因为可能有多个处理器在

并行地工作。

> **TIP** time 还提供了丰富的格式设置选项，有关这方面的大致情况，可参阅用户手册（执行命令 man time）。如果你要查看有关所有指标的摘要，可使用选项-v。

为测量基准测试程序的执行时间，Unix 命令是最简单也比较直接的方式之一。为确保测量结果是准确的，基准测试程序的执行时间应足够长（为秒级），以确保创建和删除进程的时间相比于应用程序的执行时间来说很短。指标 user 适合用于监视 CPU 的性能，而指标 real 也包含等待输入/输出操作期间用在其他进程上的时间。

为了测量 Python 脚本的执行时间，另一种方便的方式是使用模块 timeit。这个模块在循环中运行代码片段 *n* 次，并测量总执行时间，然后重复这种操作 *r*（默认为 3）次，并记录其中最短的那次时间。鉴于其测量时间的方式，timeit 非常适合用于准确地测量少量语句的执行时间。

可在命令行或 IPython 中将模块 timeit 作为 Python 包执行。

IPython 是一个 Python shell，是为改善 Python 解释器的交互性而设计的。它支持按 Tab 键进行补全，还提供了很多用于对代码进行计时、剖析和调试的工具。本书自始至终都将使用这个 shell 来执行代码片段。IPython shell 支持**魔法命令**（magic command），即以符号%打头的语句，这赋予了它特殊行为。以%%打头的命名被称为**单元格魔法命令**，可应用于多行的代码片段（被称为**单元格**）。

在大多数 Linux 系统中，都可通过 pip 命令来安装 IPython。另外，Anaconda 也包含 IPython。

> **TIP** 可将 IPython 作为常规 Python shell 使用（ipython），但它还有基于 Qt 的版本（ipython qtconsole）以及使用基于浏览器的界面的版本（jupyter notebook）。

在 IPython 和命令行界面中，还可使用选项-n 和-r 分别指定循环次数和重复次数。如果没有指定，timeit 将自动推断出它们。从命令行调用 timeit 时，还可使用选项-s 传入一些设置代码——在基准测试程序之前执行的代码。下面演示了如何在 IPython 和命令行中执行 timeit。

```
# IPython 界面
$ ipython
In [1]: from simul import benchmark
In [2]: %timeit benchmark()
1 loops, best of 3: 782 ms per loop

# 命令行界面
$ python -m timeit -s 'from simul import benchmark' 'benchmark()'
10 loops, best of 3: 826 msec per loop

# Python 界面
# 将这个函数放到脚本 simul.py 中

import timeit
```

```
result = timeit.timeit('benchmark()',
 setup='from __main__ import benchmark',
 number=10)

# 结果为整个循环的执行时间（单位为秒）
result = timeit.repeat('benchmark()',
 setup='from __main__ import benchmark',
 number=10,
 repeat=3)
# 结果是一个列表，其中包含每次的执行时间（这里重复 3 次）
```

请注意，在命令行界面和 IPython 界面中，会自动确定合理的循环次数（n），但在 Python 界面中，必须通过参数 number 显式地指定循环次数。

## 1.3　使用 pytest-benchmark 编写更佳的测试和基准测试程序

Unix 命令 time 是个多功能工具，可在各种平台上用来评估小程序的执行时间。对于较大的 Python 应用程序和库，要对其进行测试和基准测试，一种更全面的解决方案是结合使用 pytest 及其插件 pytest-benchmark。

在本节中，我们将使用测试框架 pytest 为应用程序编写一个简单的基准测试程序。如果读者想更详细地了解这个框架及其用法，pytest 文档是最佳的资源，其网址为 http://doc.pytest.org/en/latest/。

> 可在控制台中使用命令 pip install pytest 来安装 pytest，其基准测试插件也可以类似的方式安装——使用命令 pip install pytest-benchmark。

测试框架是一组测试工具，可简化编写、执行和调试测试的工作，还提供了丰富的测试结果报告和摘要。使用框架 pytest 时，建议将测试和应用程序代码放在不同的文件中。在下面的示例中，我们创建了文件 test_simul.py，其中包含函数 test_evolve。

```
from simul import Particle, ParticleSimulator

def test_evolve():
    particles = [Particle( 0.3, 0.5, +1),
                 Particle( 0.0, -0.5, -1),
                 Particle(-0.1, -0.4, +3)]

    simulator = ParticleSimulator(particles)

    simulator.evolve(0.1)

    p0, p1, p2 = particles

    def fequal(a, b, eps=1e-5):
        return abs(a - b) < eps
```

```
    assert fequal(p0.x, 0.210269)
    assert fequal(p0.y, 0.543863)

    assert fequal(p1.x, -0.099334)
    assert fequal(p1.y, -0.490034)

    assert fequal(p2.x, 0.191358)
    assert fequal(p2.y, -0.365227)
```

可在命令行中运行可执行文件 `pytest`，它将找到并运行 Python 模块中的测试。要执行特定的测试，可使用语法 `pytest path/to/module.py::function_name`。为执行 `test_evolve`，可在控制台中输入如下命令，这将获得简单但信息丰富的输出。

```
$ pytest test_simul.py::test_evolve

platform linux -- Python 3.5.2, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
rootdir: /home/gabriele/workspace/hiperf/chapter1, inifile: plugins:
collected 2 items

test_simul.py .

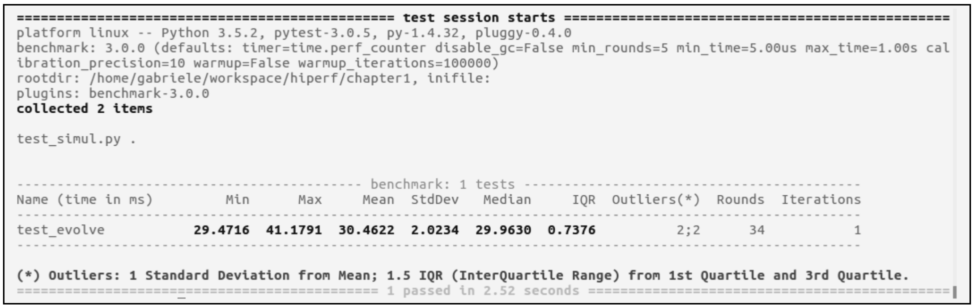
=========================== 1 passed in 0.43 seconds
===========================
```

编写好测试后，就可使用插件 `pytest-benchmark` 将测试作为基准测试程序来执行。如果我们修改函数 `test_evolve`，使其接受一个名为 `benchmark` 的参数，框架 `pytest` 将自动将资源 `benchmark` 作为参数传递给这个函数。在 `pytest` 中，这些资源被称为**测试夹具**（fixture）。为调用基准测试资源，可将要作为测试基准程序的函数作为第一个参数，并在它后面指定其他参数。下面演示了为对函数 `ParticleSimulator.evolve` 进行基准测试，需要对代码做哪些修改。

```
from simul import Particle, ParticleSimulator

def test_evolve(benchmark):
    # ……以前的代码
    benchmark(simulator.evolve, 0.1)
```

为运行基准测试，只需再次执行命令 `pytest test_simul.py::test_evolve` 即可。输出将包含有关函数 `test_evolve` 的详细计时信息，如下所示。



```
============================================= test session starts =============================================
platform linux -- Python 3.5.2, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
benchmark: 3.0.0 (defaults: timer=time.perf_counter disable_gc=False min_rounds=5 min_time=5.00us max_time=1.00s cal
ibration_precision=10 warmup=False warmup_iterations=100000)
rootdir: /home/gabriele/workspace/hiperf/chapter1, inifile:
plugins: benchmark-3.0.0
collected 2 items

test_simul.py .


------------------------------------------ benchmark: 1 tests ------------------------------------------
Name (time in ms)         Min      Max     Mean  StdDev   Median     IQR  Outliers(*)  Rounds  Iterations
---------------------------------------------------------------------------------------------------------
test_evolve           29.4716  41.1791  30.4622  2.0234  29.9630  0.7376          2;2      34           1
---------------------------------------------------------------------------------------------------------

(*) Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st Quartile and 3rd Quartile.
========================================== 1 passed in 2.52 seconds ==========================================
```

对于收集的每个测试，pytest-benchmark 都将执行基准测试函数多次，并提供有关其运行时间的统计摘要。前面的输出很有趣，因为它表明每次运行时执行时间都不同。

在这个示例中，test_evolve 中的基准测试函数运行了 34 次（见 Rounds 列），它们的执行时间为 29~41 毫秒不等（见 Min 和 Max 列），但平均值和中间值很接近，都是 30 毫秒左右，这与最短的执行时间相当接近。这个示例表明，每次运行时性能差别很大，因此使用只进行单次计时的工具（如 time）时，最好运行程序多次，并记录有代表性的结果，如最小值或中间值。

pytest-benchmark 还有很多其他的功能和选项，可用来精确地测量时间和分析结果。有关这方面的详细信息，可参阅其文档。

## 1.4　使用 cProfile 找出瓶颈

核实程序的正确性并测量其执行时间后，便可着手找出需要进行性能优化的代码片段了。与整个程序相比，这些代码的规模通常很小。

在 Python 标准库中，有两个剖析模块。

- 模块 **profile**：这个模块是完全使用 Python 编写的，给程序执行增加了很大的开销。这个模块之所以出现在标准库中，原因在于其强大的平台支持和易于扩展。
- 模块 **cProfile**：这是主要的剖析模块，其接口与 profile 相同。这个模块是使用 C 语言编写的，因此开销很小，适合用作通用剖析器。

可以三种不同的方式使用模块 cProfile：

- 在命令行中使用；
- 作为 Python 模块使用；
- 在 IPython 中使用。

无须对其源代码做任何修改，就可对现有 Python 脚本或函数执行 cProfile。要在命令行中使用 cProfile，可像下面这样做：

```
$ python -m cProfile simul.py
```

这将打印长长的输出，其中包含针对应用程序中调用的所有函数的多个指标。要按特定的指标对输出进行排序，可使用选项-s。在下面的示例中，输出是按后面将介绍的指标 tottime 排序的。

```
$ python -m cProfile -s tottime simul.py
```

要将 cProfile 生成的数据保存到输出文件中，可使用选项-o。cProfile 使用模块 stats 和其他工具能够识别的格式。下面演示了选项-o 的用法。

```
$ python -m cProfile -o prof.out simul.py
```

要将 `cProfile` 作为 Python 模块使用，必须像下面这样调用函数 `cProfile.run`。

```
from simul import benchmark
import cProfile

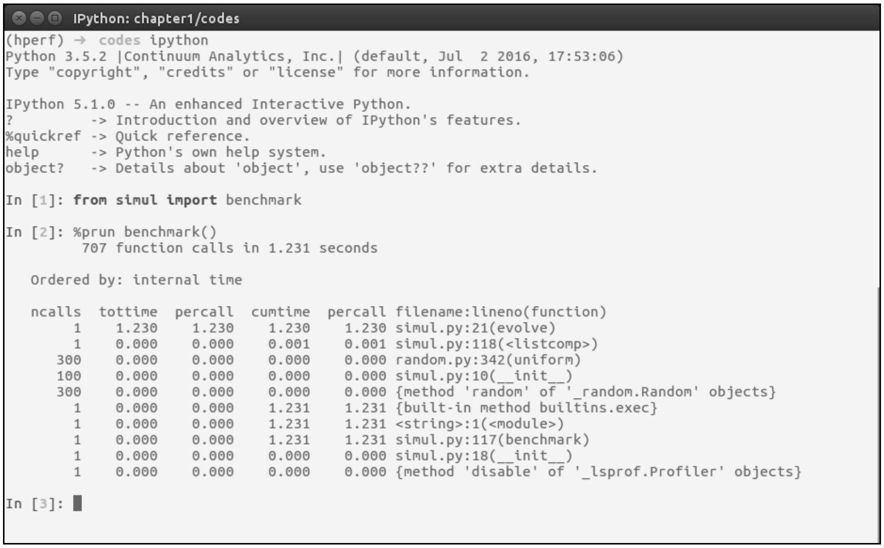
cProfile.run("benchmark()")
```

你还可在调用对象 `cProfile.Profile` 的方法的代码之间包含一段代码，如下所示。

```
from simul import benchmark
import cProfile

pr = cProfile.Profile()
pr.enable()
benchmark()
pr.disable()
pr.print_stats()
```

也可在 IPython 中以交互的方式使用 `cProfile`。魔法命令`%prun` 让你能够剖析特定的函数调用，如下图所示。




`cProfile` 的输出分成了 5 列。

- `ncalls`：函数被调用的次数。
- `tottime`：执行函数花费的总时间，不考虑其他函数调用。
- `cumtime`：执行函数花费的总时间，考虑其他函数调用。
- `percall`：单次函数调用花费的时间——可通过将总时间除以调用次数得到。

- ❑ filename:lineno：文件名和相应的行号。调用 C 语言扩展模块时，不包含这种信息。

最重要的指标是 tottime，它表示执行函数体花费的实际时间（不包含子调用），让我们能够知道瓶颈到底在哪里。

大部分时间都花在函数 evolve 上，这没什么可奇怪的。可以想见，循环是需要进行性能优化的那部分代码。

cProfile 只提供函数级信息，而不会指出导致瓶颈的具体是哪些语句。所幸工具 line_profiler 能够提供函数中各行的时间花费信息，这将在下一节介绍。

对于包含大量调用和子调用的大型程序来说，分析 cProfile 的文本输出可能是项令人望而却步的任务。有一些可视化工具可帮助完成这些任务，它们使用交互式图形界面，让你能够轻松地导航。

KCachegrind 就是一个这样的**图形用户界面**（GUI）工具，可帮助你分析 cProfile 生成的剖析输出。

> Ubuntu 16.04 官方仓库中包含 KCachegrind。Windows 用户可从 http://sourceforge.net/projects/qcachegrindwin/下载 Qt port——QCacheGrind。Mac 用户可使用 Mac Ports 编译 QCacheGrind，至于如何编译，请参阅博文“Install kcachegrind on MacOSX with ports”中的说明。

KCachegrind 无法直接读取 cProfile 生成的输出文件，所幸第三方 Python 模块 pyprof2-calltree 能够将 cProfile 输出文件转换为 KCachegrind 能够读取的格式。

> 要安装 Python Package Index 中的 pyprof2calltree，可使用命令 pip install pyprof2calltree。

为最大限度地展示 KCachegrind 的功能，我们将使用另一个结构更为多样化的示例。我们定义一个递归函数 factorial，还有另外两个使用 factorial 的函数——taylor_exp 和 taylor_sin，它们分别计算 exp(x) 和 sin(x) 的泰勒展开式的多项式系数。

```
def factorial(n):
    if n == 0:
        return 1.0
    else:
        return n * factorial(n-1)

def taylor_exp(n):
    return [1.0/factorial(i) for i in range(n)]

def taylor_sin(n):
    res = []
```

```
    for i in range(n):
        if i % 2 == 1:
            res.append((-1)**((i-1)/2)/float(factorial(i)))
        else:
            res.append(0.0)
    return res

def benchmark():
    taylor_exp(500)
    taylor_sin(500)

if __name__ == '__main__':
    benchmark()
```

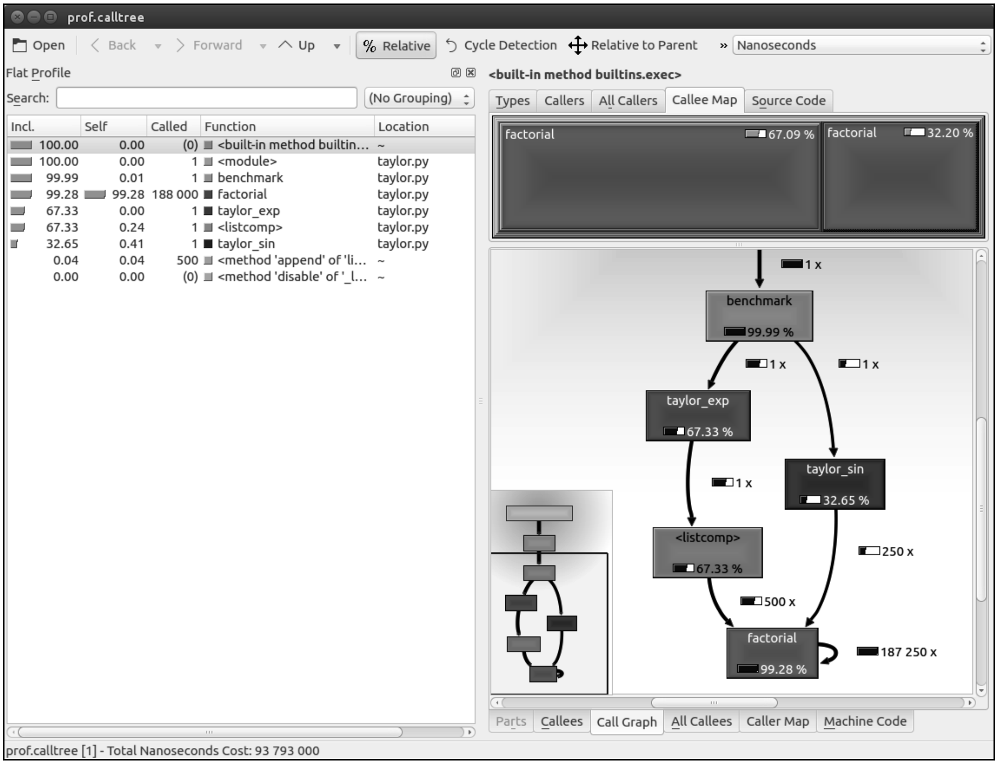
为访问剖析信息，首先需要生成 `cProfile` 输出文件。

```
$ python -m cProfile -o prof.out taylor.py
```

然后，就可使用 `pyprof2calltree` 对输出文件进行转换并启动 KCachegrind。

```
$ pyprof2calltree -i prof.out -o prof.calltree
$ kcachegrind prof.calltree # 或使用命令 qcachegrind prof.calltree
```

输出如下面的屏幕截图所示。



该屏幕截图显示了 KCachegrind 的用户界面。左边的输出与 cProfile 的输出很像，但列名稍有不同：Incl 对应于 cProfile 模块中的 cumtime，而 Self 对应于 tottime。如果你单击菜单栏中的 Relative 按钮，将以百分比的方式显示值。通过单击列名，可按相应的属性进行排序。

在右上方，如果你单击标签 Callee Map，将显示一个函数开销图。在该图中，函数占用的时间百分比与矩形面积成正比。矩形可包含子矩形，而这些子矩形表示对其他函数的子调用。在这个示例中，很容易看出有两个表示函数 factorial 的矩形，其中左边那个对应于 taylor_exp 调用的 factorial，而右边那个对应于 taylor_sin 调用的 factorial。

你可单击右边底部的标签 Call Graph 来显示另一个图——**调用图**。调用图是函数间调用关系的图形化表示，其中每个矩形都表示一个函数，而箭头表示调用关系。例如，taylor_exp 调用了 factorial 500 次，而 taylor_sin 调用了 factorial 250 次。KCachegrind 还能够发现递归调用：factorial 调用了自己 187 250 次。

你可双击矩形来切换到 Call Graph 或 Caller Map 选项卡，在这种情况下，界面将相应地更新，指出计时属性是相对于选定函数的。例如，双击 taylor_exp 将导致图形发生变化，只显示 taylor_exp 对总开销的贡献。

> 另一个用于生成调用图的流行工具是 Gprof2Dot。使用支持的剖析器生成的输出文件启动时，Gprof2Dot 将生成一个表示调用图的.dot 图。

## 1.5　使用 line_profiler 逐行进行剖析

知道哪个函数需要优化后，就可使用模块 line_profiler 来提供有关时间是如何在各行之间分配的信息。在难以确定哪些语句最费时时，这很有用。line_profiler 是 Python Package Index 提供的一个第三方模块，其安装说明请参阅 https://github.com/rkern/line_profiler。

要使用 line_profiler，需要对要监视的函数应用装饰器@profile。请注意，无须从其他模块中导入函数 profile，因为运行剖析脚本 kernprof.py 时，它将被注入全局命名空间。要对我们的程序进行剖析，需要给函数 evolve 添加装饰器@profile。

```
@profile
def evolve(self, dt):
    # 代码
```

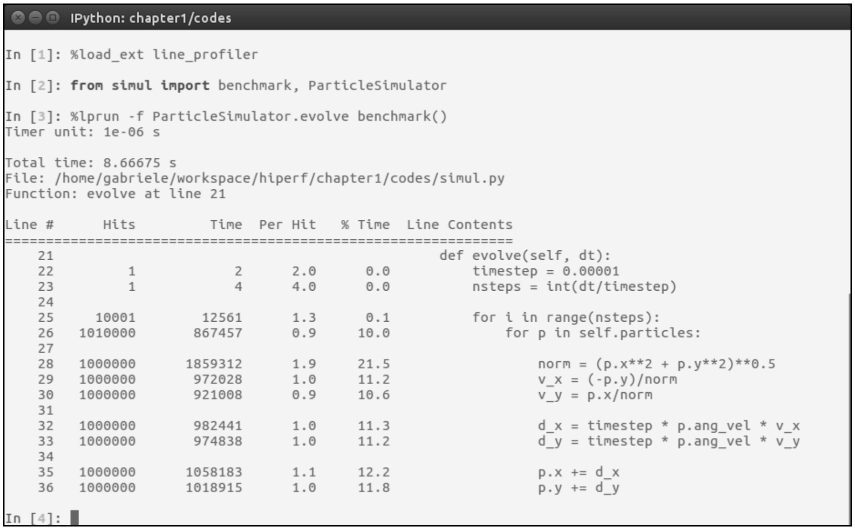
脚本 kernprof.py 生成一个输出文件，并将剖析结果打印到标准输出。运行这个脚本时，应指定两个选项。

- -l：以使用函数 line_profiler
- -v：以立即将结果打印到屏幕

下面演示了 kernprof.py 的用法：

```
$ kernprof.py -l -v simul.py
```

也可在 IPython shell 中运行这个剖析器，这样可以进行交互式编辑。你应首先加载 `line_profiler` 扩展，它提供了魔法命令 `lprun`。使用这个命令，就无须添加装饰器`@profile`。




输出非常直观，分成了 6 列。

- `Line #`：运行的代码行号。
- `Hits`：代码行运行的次数。
- `Time`：代码行的执行时间，单位为微秒。
- `Per Hit`：Time/Hits。
- `% Time`：代码行总执行时间所占的百分比。
- `Line Contents`：代码行的内容。

只需查看`% Time`列，就可清楚地知道时间都花在了什么地方。在这个示例中，`for`循环中几条语句占用的时间百分比都在 10%~20%。

## 1.6 优化代码

确定应用程序的大部分时间都花在什么地方后，就可做些修改，并评估修改对性能的影响。

要优化纯粹的 Python 代码，方式有多种，其中效果最显著的方式是对使用的**算法**进行改进。就这个示例而言，相比于计算速度并逐步累积位移，效率更高（而且绝对准确而不是近似）的方

式是，使用半径（r）和角度（alpha）来表示运动方程，再使用下面的方程来计算粒子在圆周上的位置。

```
x = r * cos(alpha)
y = r * sin(alpha)
```

另一种方式是最大限度地减少指令数。例如，可预先计算不随时间变换的因子 timestep * p.ang_vel factor。为此，我们可交换循环顺序（先迭代粒子，再迭代时段），并将计算前述因子的代码放在迭代粒子的循环外面。

逐行剖析结果还表明，即便是简单的赋值操作，也可能消耗大量的时间。例如，下面的语句占用的时间超过了总时间的 10%。

```
v_x = (-p.y)/norm
```

可通过减少赋值操作数量来改善这个循环的性能，为此可将这个表达式改写为单条更复杂些的语句，以消除中间变量（请注意，将先计算等号右边的部分，再将结果赋给变量）。

```
p.x, p.y = p.x - t_x_ang*p.y/norm, p.y + t_x_ang * p.x/norm
```

这样修改后的代码如下所示。

```
    def evolve_fast(self, dt):
        timestep = 0.00001
        nsteps = int(dt/timestep)

        # 调整了循环顺序
        for p in self.particles:
            t_x_ang = timestep * p.ang_vel
            for i in range(nsteps):
                norm = (p.x**2 + p.y**2)**0.5
                p.x, p.y = (p.x - t_x_ang * p.y/norm,
                            p.y + t_x_ang * p.x/norm)
```

修改代码后，应运行测试以核实结果与以前相同，再使用基准测试对执行时间进行比较。

```
$ time python simul.py # 优化性能后
real    0m0.756s
user    0m0.714s
sys     0m0.036s

$ time python simul.py # 原来的代码
real    0m0.863s
user    0m0.831s
sys     0m0.028s
```

如你所见，进行纯粹而细微的 Python 优化后，速度有所提高，但并不显著。

## 1.7 模块 dis

在某些情况下，要估计 Python 语句将执行多少操作并不容易。在本节中，我们将深入 Python 内部，以估计各条语句的性能。在 CPython 解释器中，Python 代码首先被转换为中间表示——**字节码**，再由 Python 解释器执行。

要了解代码是如何转换为字节码的，可使用 Python 模块 `dis`（dis 表示 disassemble，即反汇编）。这个模块的用法非常简单，只需对目标代码（这里是方法 `ParticleSimulator.evolve`）调用函数 `dis.dis` 即可。

```
import dis
from simul import ParticleSimulator
dis.dis(ParticleSimulator.evolve)
```

这将打印每行代码对应的字节码指令列表。例如，语句 `v_x = (-p.y) /norm` 被转换为下面一组指令。

```
29          85 LOAD_FAST                5 (p)
            88 LOAD_ATTR                4 (y)
            91 UNARY_NEGATIVE
            92 LOAD_FAST                6 (norm)
            95 BINARY_TRUE_DIVIDE
            96 STORE_FAST               7 (v_x)
```

其中 `LOAD_FAST` 将指向变量 `p` 的引用加载到栈中，而 `LOAD_ATTR` 加载栈顶元素的属性 `y`。其他指令（`UNARY_NEGATIVE` 和 `BINARY_TRUE_DIVIDE`）只是对栈顶元素执行算术运算。最后，结果被存储到 `v_x` 中（`STORE_FAST`）。

通过分析 `dis` 的输出可知，循环的第一个版本被转换为 51 个字节码指令，而第二个版本被转换为 35 个指令。

模块 `dis` 能够让你知道语句是如何被转换的，但主要用作探索和学习 Python 字节码的工具。

要进一步改善性能，可继续尝试找出其他减少指令数量的方法。然而，这种方法显然最终受制于 Python 解释器的速度，而对有些工作来说，Python 解释器可能不是合适的工具。在后续章节中，我们将介绍如何提高那些受制于解释器的计算的速度——执行使用 C 或 Fortan 等低级语言编写的快速专用版本。

## 1.8 使用 memory_profiler 剖析内存使用情况

在有些情况下，消耗大量的内存是个问题。例如，如果我们要处理大量的粒子，就需要创建大量的 `Particle` 实例，这将带来很大的内存开销。

模块 memory_profiler 以类似于 line_profiler 的方式，提供有关进程内存使用情况的摘要。

> Python Package Index 也提供了 memory_profiler 包。你应同时安装模块 psutil，这样将极大地提高 memory_profiler 的速度。

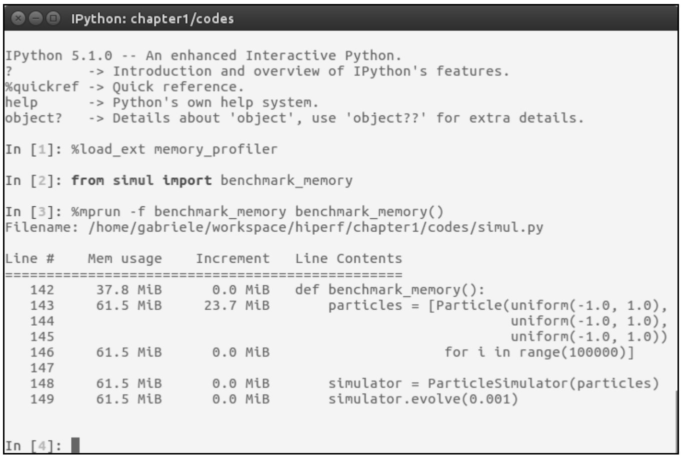
与 line_profiler 一样，memory_profiler 也要求对源代码进行处理：给要监视的函数加上装饰器@profile。在这个示例中，我们要分析的是函数 benchmark。

我们可稍微修改函数 benchmark，以实例化大量（100 000 个）Particle 实例，并缩短模拟时间。

```
def benchmark_memory():
    particles = [Particle(uniform(-1.0, 1.0),
                          uniform(-1.0, 1.0),
                          uniform(-1.0, 1.0))
                 for i in range(100000)]

    simulator = ParticleSimulator(particles)
    simulator.evolve(0.001)
```

我们可在 IPython shell 中使用 memory_profiler，为此可使用魔法命令%mprun，如下面的屏幕截图所示。




> 在 IPython shell 中，也可使用命令 mprof run 来运行 memory_profiler，但这样做之前必须先给要监视的函数添加@profile 装饰器。

从 Increment 列可知，100 000 个 Particle 对象占用了 23.7MiB 的内存。

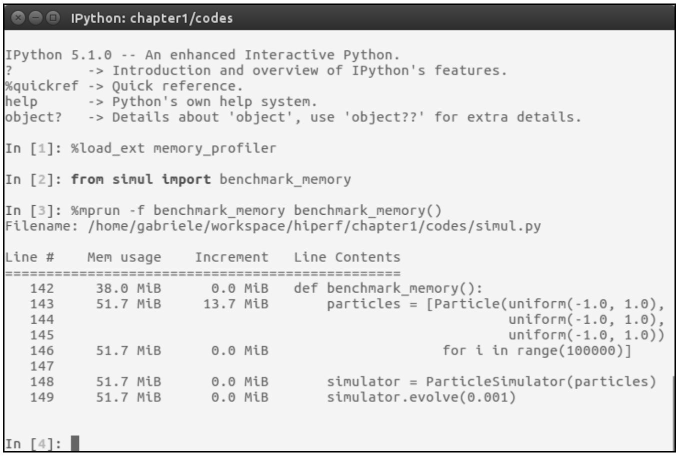
1MiB（兆字节）相当于 1 048 576 字节，这不同于 1MB（百万字节），后者相当于 1 000 000 字节。

为减少内存消耗，可在 `Particle` 类中使用`__slots__`。这将避免将实例的变量存储在内部字典中，从而节省一些内存。然而，这种策略也有缺点：不能添加`__slots__`中没有指定的属性。

```
class Particle:
    __slots__ = ('x', 'y', 'ang_vel')

    def __init__(self, x, y, ang_vel):
        self.x = x
        self.y = y
        self.ang_vel = ang_vel
```

现在可以再次运行基准测试，以评估内存消耗的变化情况，结果如下面的屏幕截图所示。




通过使用`__slots__`重写 `Particle` 类，节省了大约 10MiB 内存。

## 1.9 小结

本章介绍了基本的优化原则，并将这些原则应用于一个测试应用程序。优化时，首先要做的是测试，并找出应用程序的瓶颈。你学会了如何编写基准测试程序，以及如何使用 Unix 命令 `time`、Python 模块 `timeit` 和功能齐备的 `pytest-benchmark` 包来测量基准测试程序的执行时间。你还学习了如何使用 `cProfile`、`line_profiler` 和 `memory_profiler` 对应用程序进行剖析，以及如何使用 KCachegrind 以图形化方式分析和导航剖析数据。

下一章将探索如何使用 Python 标准库中的算法和数据结构来改善性能，你将学习可伸缩性（scaling）、多个数据结构的用法以及缓存和 memoization 等技巧。

第 2 章

# 纯粹的 Python 优化

前一章说过，要改善应用程序的性能，最有效的方式之一是使用更合适的算法和数据结构。Python 标准库提供了大量现成的算法和数据结构，你可在应用程序中直接使用它们。有了本章介绍的工具，你就能够使用合适的算法来完成任务，从而极大地提升速度。

虽然很多算法历史悠久，但它们在当今世界依然适用，这是因为我们在不断地生成、使用和分析越来越多的数据。在有些情况下，购买规模更大的服务器或进行微优化行之有效，但通过改进算法来增强可伸缩性可一劳永逸地解决问题。

本章将介绍如何使用标准算法和数据结构来增强可伸缩性，并利用第三方库阐述更复杂的用例。本章还将介绍实现缓存的工具，缓存是一种以内存或磁盘空间换取响应时间的技巧。

本章将介绍的主题如下：

- 计算复杂性；
- 列表和队列；
- 字典；
- 如何使用字典创建反向索引；
- 集；
- 堆和优先队列；
- 使用字典树（trie）实现自动补全；
- 缓存；
- 使用装饰器 `functools.lru_cache` 实现内存缓存；
- 使用 `joblib.Memory` 实现磁盘缓存；
- 使用推导和生成器实现速度快且占用内存少的循环。

## 2.1 有用的算法和数据结构

对提升性能而言，改进算法特别有效，因为这通常可增强应用程序的可伸缩性，从而能够处

理更多的输入。

计算复杂性是一个描述执行任务所需资源的指标，可根据它对算法进行分类。这样的分类是使用大 O 表示法来表示的。所谓大 O 表示法，指的是为完成任务需要执行的操作数的上限，这通常取决于输入的规模。



例如，要将列表的每个元素都加 1，可像下面这样使用一个 `for` 循环来实现：

```
input = list(range(10))
for i, _ in enumerate(input):
    input[i] += 1
```

如果操作不依赖于输入的规模（如访问列表的第一个元素），相应算法所需的时间就被认为是固定的，用 $O(1)$表示。这意味着不管有多少数据，运行算法所需的时间都相同。

在上述简单的算法中，操作 `input[i] += 1` 将重复 10 次，这与输入的规模相同。如果将输入的规模翻倍，操作数将成比例地增加。由于操作数与输入规模成正比，这种算法所需的时间为 $O(N)$，其中 $N$ 为输入的规模。

在有些情况下，运行时间可能取决于输入的结构，如集合是否是有序的，以及是否包含很多重复的元素。在这些情况下，算法的最佳、平均和最糟运行时间可能不同。除非特别指出了，否则本章所说的运行时间都是指平均运行时间。

在本节中，我们将研究主要算法的运行时间以及 Python 标准库实现的主要数据结构，并将了解到缩短运行时间好处多多，让我们能够优雅地解决规模庞大的问题。

本章用来运行基准测试的代码可在笔记本（notebook）`Algorithms.ipynb` 中找到，你可使用 Jupyter 来打开它。

### 2.1.1　列表和双端队列

Python 列表是有序的元素集合，在 Python 中是使用大小可调整的数组实现的。数组是一种基本数据结构，由一系列连续的内存单元组成，其中每个内存单元都包含指向一个 Python 对象的引用。

在访问、修改和附加元素方面，列表表现得非常出色。要访问或修改元素，需要从底层数组的相应位置获取对象引用，因此其复杂度为 $O(1)$。附加元素的速度也非常快。当你创建一个空列表时，将分配一个长度固定的数组；而当你插入元素时，数组中的位置将逐渐被填满。当所有位置都被占据后，列表需要增大其底层数组的长度，进而触发内存重新分配，这需要的时间为 $O(N)$。尽管如此，内存分配操作并不频繁，因此附加操作的时间复杂度接近于 $O(1)$。

在列表开头（或中间）添加或删除元素的操作可能在效率方面存在问题。在列表开头插入或

删除元素时，后续所有元素都需要移动一个位置，因此需要的时间为 $O(N)$。

下表列出了对包含不同数量的元素的列表执行各种操作所需的时间。从中可知，在列表开头和末尾插入和删除元素时，性能方面的差别非常大。

| 代 码 | $N$=10 000 (μs) | $N$=20 000 (μs) | $N$=30 000 (μs) | 时 间 |
|---|---|---|---|---|
| `list.pop()` | 0.50 | 0.59 | 0.58 | $O(1)$ |
| `list.pop(0)` | 4.20 | 8.36 | 12.09 | $O(N)$ |
| `list.append(1)` | 0.43 | 0.45 | 0.46 | $O(1)$ |
| `list.insert(0, 1)` | 6.20 | 11.97 | 17.41 | $O(N)$ |

在有些情况下，必须高效地执行在集合开头和末尾插入或删除元素的操作，Python 通过 `collections.deque` 类提供了一种具有这种特征的数据结构。`deque` 指的是双端队列，因为这种数据结构被设计成能够在集合两端高效地添加和删除元素，就像在队列中执行这些操作一样。在 Python 中，双端队列是以双向链表的方式实现的。

除 `pop` 和 `append` 外，双端队列还暴露了方法 `popleft` 和 `appendleft`，它们的运行时间都是 $O(1)$。

| 代 码 | $N$=10 000 (μs) | $N$=20 000 (μs) | $N$=30 000 (μs) | 时 间 |
|---|---|---|---|---|
| `deque.pop()` | 0.41 | 0.47 | 0.51 | $O(1)$ |
| `deque.popleft()` | 0.39 | 0.51 | 0.47 | $O(1)$ |
| `deque.append(1)` | 0.42 | 0.48 | 0.50 | $O(1)$ |
| `deque.appendleft(1)` | 0.38 | 0.47 | 0.51 | $O(1)$ |

虽然双端队列有这些优点，但在大多数情况下，都不应用它来替换常规列表。方法 `appendleft` 和 `popleft` 的高效是要付出代价的：访问双端队列中间的元素所需的时间为 $O(N)$，如下表所示。

| 代 码 | $N$=10 000 (μs) | $N$=20 000 (μs) | $N$=30 000 (μs) | 时 间 |
|---|---|---|---|---|
| `deque[0]` | 0.37 | 0.41 | 0.45 | $O(1)$ |
| `deque[N - 1]` | 0.37 | 0.42 | 0.43 | $O(1)$ |
| `deque[int(N / 2)]` | 1.14 | 1.71 | 2.48 | $O(N)$ |

在列表中查找元素通常是一种 $O(N)$操作，这种操作是使用方法 `list.index` 来完成的。为提高列表查找速度，一种简单的办法是确保底层数组是有序的，并使用模块 `bisect` 来执行二分查找。

模块 `bisect` 让你能够在有序数组中进行快速查找。对于有序列表，可使用函数 `bisect.bisect` 来确定将元素插入到什么位置，同时可确保插入后列表依然是有序的。从下面的示例可知，要在列表中插入元素 3，并确保插入后列表依然是有序的，应将元素 3 放在第三个位置（对应的索引为 2）。

```
insert bisect
collection = [1, 2, 4, 5, 6]
bisect.bisect(collection, 3)
# 结果：2
```

这个函数使用二分查找算法，运行时间为 $O(\log(N))$。这样的运行速度非常快，大致意味着输入规模每翻一倍，运行时间都只会增加固定的量。这意味着如果程序在输入规模为 1000 时执行时间为 1 秒，则处理规模为 2000 的输入需要 2 秒，处理规模为 4000 的输入需要 3 秒，以此类推。如果有 100 秒时间可用，从理论上说就可处理规模为 $10^{33}$ 的输入，这比你身体包含的原子数还多！

如果要插入的值已包含在列表中，函数 `bisect.bisect` 将返回这个既有值**后面**的位置。因此，我们可使用变种 `bisect.bisect_left`，它以下面这样的方式返回正确的索引。

```
def index_bisect(a, x):
  '找到第一个与 x 相同的值'
  i = bisect.bisect_left(a, x)
  if i != len(a) and a[i] == x:
  return i
  raise ValueError
```

从下表可知，使用 `bisect` 的解决方案的运行时间几乎不受输入规模的影响，适合用来搜索非常大的集合。

| 代　　码 | *N*=10 000 (μs) | *N*=20 000 (μs) | *N*=30 000 (μs) | 时　　间 |
|---|---|---|---|---|
| `list.index(a)` | 87.55 | 171.06 | 263.17 | $O(N)$ |
| `index_bisect(list, a)` | 3.16 | 3.20 | 4.71 | $O(\log(N))$ |

## 2.1.2　字典

字典功能丰富，在 Python 语言中被广泛使用。字典是以散列映射的方式实现的，在插入、删除和访问元素方面的表现都非常杰出——所有这些操作的时间复杂度都是 $O(1)$。

> 在 Python 3.5 及以前的版本中，字段是无序集合，但从 Python 3.6 起，字典能够保留元素的插入顺序。

散列映射是一种将键关联到值的数据结构，其背后的原理是给每个键都指定索引，以便将关联的值存储在数组中。索引可使用散列函数计算得到；Python 为多种数据类型实现了散列函数，例如，获取散列码的通用函数是 `hash`。下面的示例演示了如何在给定字符串`"hello"`的情况下获取散列码。

```
hash("hello")
# 结果：-1182655621190490452

# 要将得到的数字限制在特定范围内
```

```
# 可使用求模运算符 (%)
hash("hello") % 10
# 结果：8
```

散列映射实现起来可能比较棘手，因为它们需要处理冲突，即两个不同对象的散列码相同。然而，所有的复杂性都被隐藏在实现后面，且在大多数情况下，默认的冲突解决方案就挺管用。

不管字典的规模如何，访问、插入和删除元素的运行时间都为 $O(1)$。然而，别忘了还需计算散列函数，而就字符串而言，这种计算所需的时间与字符串长度成正比。考虑到字符串键通常较短，因此在实践中这不是问题。

使用字典可高效地计算列表中独特元素的个数。在下面的示例中，我们定义了函数 counter_dict，它接受一个列表并返回一个字典，其中包含列表中每个独特值的出现次数。

```
def counter_dict(items):
    counter = {}
    for item in items:
        if item not in counter:
            counter[item] = 0
        else:
            counter[item] += 1
    return counter
```

collections.defaultdict 生成一个字典，并给每个新键自动指定一个默认值，通过使用它可在一定程度上简化上述代码。在下面的代码中，调用 defaultdict(int)生成一个字典，其中每个新键都被自动指定为零值，因此可使用它来简化计数工作。

```
from collections import defaultdict
def counter_defaultdict(items):
    counter = defaultdict(int)
    for item in items:
        counter[item] += 1
    return counter
```

模块 collections 还包含一个名为 Counter 的类，可用来实现同样的目的，但只需一行代码。

```
from collections import Counter
counter = Counter(items)
```

在速度方面，这些计数方式的时间复杂度都相同，但使用 Counter 实现的效率最高，如下表所示。

| 代　　码 | $N$=1000 (μs) | $N$=2000 (μs) | $N$=3000 (μs) | 时　　间 |
|---|---|---|---|---|
| Counter(items) | 51.48 | 96.63 | 140.26 | $O(N)$ |
| counter_dict(items) | 111.96 | 197.13 | 282.79 | $O(N)$ |
| counter_defaultdict(items) | 120.90 | 238.27 | 359.60 | $O(N)$ |

### 使用散列映射在内存中创建查找索引

使用字典可在文档列表中快速查找特定的单词，就像搜索引擎一样。在这一小节中，你将学习如何创建基于列表字典的反向索引。假设有一个包含 4 个文档的集合：

```
docs = ["the cat is under the table",
        "the dog is under the table",
        "cats and dogs smell roses",
        "Carla eats an apple"]
```

要获取与查询匹配的所有文档，一种简单的方式是扫描每个文档，并检查其中是否包含指定的单词。例如，要查找包含单词 table 的文档，可使用如下过滤操作。

```
matches = [doc for doc in docs if "table" in doc]
```

这种方法很简单，对一次性查询来说也挺管用，但如果需要频繁地查询这个集合，对查询时间进行优化将大有裨益。由于线性扫描的总查询开销为 $O(N)$，因此可以想见，提高可伸缩性后，将能够处理大得多的文档集合。

一种更佳的策略是花些时间对文档进行预处理，以便查询时更容易找到它们。我们可创建一个名为**反向索引**的结构，它将集合中的每个单词都关联到包含该单词的文档列表。在前面的示例中，单词 table 将关联到文档 the cat is under the table 和 the dog is under the table，而这两个文档的索引分别为 0 和 1。

为实现这种映射，可遍历文档集合，并将包含指定单词的文档的索引存储在一个字典中。这种实现与函数 `counter_dict` 类似，但不累积计数器，而是不断增大列表，其中包含与指定单词匹配的文档。

```
# 创建索引
index = {}
for i, doc in enumerate(docs):
    # 遍历文档中的每个单词
    for word in doc.split():
        # 创建一个列表，其中包含所有包含指定单词的文档的索引
        if word not in index:
            index[word] = [i]
        else:
            index[word].append(i)
```

创建索引后，查询时只需执行简单的字典查找。例如，要返回所有包含单词 table 的文档，只需查询索引，并获取相应的文档。

```
results = index["table"]
result_documents = [docs[i] for i in results]
```

有了索引后，查询集合时只需执行一次字典访问操作，因此查询的时间复杂度为 $O(1)$！多亏了反向索引，现在无论查询多少文档（只要它们都能够加入到内存中），所需的时间都一样。毋

庸置疑，索引技术不仅被搜索引擎广泛用来快速检索数据，还被数据库以及其他所有需要快速搜索的系统所使用。

请注意，创建反向索引是一种代价高昂的操作，必须考虑到每个可能的查询。这是一个严重的缺点，但好处非常大，值得为此付出灵活性降低的代价。

### 2.1.3　集

集是一个无序的元素集合，且其中的每个元素都必须是独一无二的。集的主要用途是成员资格测试（检查集合中是否包含特定的元素），集操作包含并集、差集和交集。

在 Python 中，集与字典一样，也是使用基于散列的算法实现的，因此其加法、删除和成员资格测试等操作的时间复杂度都为 $O(1)$，即不受集合规模的影响。

集中的元素都是独一无二的，因此其一种显而易见的用途是用于删除集合中重复的元素，为此只需将集合传递给构造函数 `set` 即可，如下所示。

```
# 创建一个包含重复元素的列表
x = list(range(1000)) + list(range(500))
# 集 x_unique 将只包含 x 中不同的元素
x_unique = set(x)
```

删除重复元素的时间复杂度为 $O(N)$，因为这种操作要求读取输入，并将每个不同的元素加入到集中。

集支持大量的操作，如并集、交集和差集。并集包含两个集中所有的元素；交集包含两个集中都有的元素；而差集包含出现在第一个集中但没有出现在第二个集中的元素。这些操作的时间复杂度如下表所示。请注意，由于这些操作涉及两个规模不同的集，因此我们用 $S$ 表示第一个集（`s`）的规模，并用 $T$ 表示第二个集（`t`）的规模。

| 代　　码 | 时　　间 |
|---|---|
| `s.union(t)` | $O(S+T)$ |
| `s.intersection(t)` | $O(\min(S, T))$ |
| `s.difference(t)` | $O(S)$ |

集操作的一种用途是布尔查询。在前面的反向索引示例中，你可能想支持包含多个单词的查询。例如，你可能想查找所有包含单词 cat 和 table 的文档。要高效地执行这种查询，可计算包含单词 cat 的文档集和包含单词 table 的文档集的交集。

为高效地支持这种操作，可修改前面创建索引的代码，将每个单词都关联到一个文档集（而不是文档列表）。这样修改后，只需执行合适的集操作就能完成更复杂的查询。在下面的代码中，反向索引是基于集的，而查询是使用集操作来完成的。

```
# 使用集来创建索引
index = {}
for i, doc in enumerate(docs):
    # 遍历文档中的每个单词
    for word in doc.split():
        # 创建一个集，其中包含出现了指定单词的所有文档的索引
        if word not in index:
            index[word] = {i}
        else:
            index[word].add(i)

# 查询包含单词cat 和 table 的文档
```

2

### 2.1.4　堆

堆是一种设计用于快速查找并提取集合中最大值或最小值的数据结构，其典型用途是按优先级处理一系列任务。

从理论上说，可结合使用有序列表和模块 `bisect` 中的工具来替代堆，这样提取最大值的时间复杂度将为 $O(1)$（使用 `list.pop`），但插入操作的时间复杂度仍为 $O(N)$（别忘了，虽然找出插入位置的时间复杂度为 $O(\log(N))$，但在列表中间插入元素的时间复杂度仍为 $O(N)$）。堆是一种效率更高的数据结构，其元素插入操作和最大值提取操作的时间复杂度都为 $O(\log(N))$。

在 Python 中，堆是通过对列表执行模块 `heapq` 中的函数来创建的。例如，如果有一个包含10个元素的列表，可使用函数 `heapq.heapify` 将其转换为堆。

```
import heapq

collection = [10, 3, 3, 4, 5, 6]
heapq.heapify(collection)
```

要对堆执行插入和提取操作，可使用函数 `heapq.heappush` 和 `heapq.heappop`。函数 `heapq.heappop` 提取集合中的最小值，时间复杂度为 $O(\log(N))$，其用法如下。

```
heapq.heappop(collection)
# 返回：3
```

同理，要压入整数 1，可使用函数 `heapq.heappush`，如下所示。

```
heapq.heappush(collection, 1)
```

另一种方法是使用 `queue.PriorityQueue` 类，它还是线程和进程安全的。要在 `PriorityQueue` 类中填充元素，可使用方法 `PriorityQueue.put`；要提取最小值，可使用方法 `PriorityQueue.get`。

```
from queue import PriorityQueue

queue = PriorityQueue()
for element in collection:
```

```
    queue.put(element)

queue.get()
# 返回：3
```

要提取最大的元素，可采用一种简单的诀窍——将每个元素都乘以–1，这将反转元素的排列顺序。另外，如果要将每个数字（可能表示优先级）关联到一个对象（如要执行的任务），可插入形如(number, object)的元组，这是因为元组的比较运算符将根据其第一个元素进行排序，如下面的示例所示。

```
queue = PriorityQueue()
queue.put((3, "priority 3"))
queue.put((2, "priority 2"))
queue.put((1, "priority 1"))
queue.get()
# 返回：(1, "priority 1")
```

## 2.1.5　字典树

字典树也被称为前缀树，这种数据结构可能不那么流行，但很有用。在列表中查找与前缀匹配的字符串方面，字典树的速度极快，因此非常适合用来实现输入时查找和自动补全功能。实现自动补全功能时，候选内容列表非常大，因此要求响应时间很短。

遗憾的是，Python 标准库没有提供字典树实现，但通过 PyPI 可找到很多高效的实现。本节将使用的实现是 patricia-trie，它只包含一个文件，而且完全是使用 Python 编写的。例如，我们将使用 patricia-trie 在一系列字符串中找出最长前缀（就像自动补全那样）。

例如，我们可演示在搜索字符串列表方面，字典树的搜索速度有多快。为生成大量各不相同的随机字符串，我们可定义一个函数——random_string。这个函数返回一个由随机大写字符组成的字符串，虽然这可能生成重复的字符串，但只要让字符串足够长，就可将这种可能性降低到可忽略不计的程度。函数 random_string 的实现如下：

```
from random import choice
from string import ascii_uppercase

def random_string(length):
  """生成一个由 length 个大写 ASCII 字符组成的随机字符串"""
  return ''.join(choice(ascii_uppercase) for i in range(length))
```

我们可创建一个随机字符串列表，使用函数 str.startswith 在其中搜索前缀（这里为字符串 AA），并看看这种操作的速度有多快。

```
strings = [random_string(32) for i in range(10000)]
matches = [s for s in strings if s.startswith('AA')]
```

列表推导和 str.startwith 都是经过极度优化的操作，在这个很小的数据集中搜索时，只需要大约 1 毫秒。

```
%timeit [s for s in strings if s.startswith('AA')]

1000 loops, best of 3: 1.76 ms per loop
```



下面尝试使用前缀字典来执行这种操作。在这个示例中，我们都将使用 patricia-trie 库，你可使用 pip 来安装它。patricia.trie 类实现了字典树数据结构的一个变种，并提供了类似于字典的接口。为初始化字典树，可先根据字符串列表创建一个字典，如下所示。

```
from patricia import trie
strings_dict = {s:0 for s in strings}
# 一个所有值都为 0 的字典
strings_trie = trie(**strings_dict)
```

要查询与前缀匹配的内容，可使用方法 trie.iter，它返回一个迭代器，该迭代器可用于迭代匹配的字符串。

```
matches = list(strings_trie.iter('AA'))
```

知道如何初始化和查询字典树后，就可计算操作的时间了。

```
%timeit list(strings_trie.iter('AA'))
10000 loops, best of 3: 60.1 µs per loop
```

如果你仔细查看，将发现在前述输入规模下，操作的执行时间为 60.1 微秒，比线性搜索的速度快了大约 30 倍（1.76 毫秒=1760 微秒）！速度之所以有如此惊人的提高，是因为字典树前缀搜索的计算复杂度更低。字典树查询的时间复杂度为 $O(S)$，其中 $S$ 为集合中最长的字符串的长度，而简单线性扫描的时间复杂度为 $O(N)$，其中 $N$ 为集合的长度。

请注意，如果要返回所有与前缀匹配的字符串，运行时间将与跟前缀匹配的字符串数量成正比。因此，设计基准测试程序时，必须特别小心，确保每次返回的结果数都相同。

下表比较了字典树和线性扫描的可伸缩性，其中涉及的数据集的规模各不相同，但返回的前缀匹配结果都是 10 个。

| 算　法 | $N$=10 000 (µs) | $N$=20 000 (µs) | $N$=30 000 (µs) | 时　间 |
|---|---|---|---|---|
| 字典树 | 17.12 | 17.27 | 17.47 | $O(S)$ |
| 线性扫描 | 1978.44 | 4075.72 | 6398.06 | $O(N)$ |

有趣的是，patricia-trie 的实现实际上只包含一个 Python 文件，这清楚地表明，巧妙的算法既简单又强大。要获得其他功能并进一步提高性能，可使用采用 C 语言编写并经过优化的字典树库，如 datrie 和 marisa-trie。

## 2.2　缓存和 memoization

缓存是一种出色的技术，用于改善各种应用程序的性能，其背后的理念是将好不容易得到的结果存储在临时区域。这种区域被称为缓存区，可以是内存、磁盘或远程位置。

Web 应用大量地使用了缓存技术。在 Web 应用中，常常会发生多位用户同时请求同一个页面的情况。在这种情况下，Web 应用可只生成网页一次，并向用户提供已渲染好的页面，而不是在每位用户请求时都重复生成页面。理想情况下，缓存技术还需使用有效的验证机制，以便需要更新网页时重新生成，再将其提供给用户。智能缓存技术让 Web 应用能够处理更多的用户，同时消耗更少的资源。你还可预先进行缓存，例如，在用户在线观看视频时，缓存视频的后续部分。

对于有些算法，还可使用缓存技术来改善其性能，一个典型示例是计算斐波纳契数列。由于计算斐波纳契数列中的下一个数字时，需要用到前一个数字，因此可存储并重用以前的结果，从而极大地缩短运行时间。在应用程序中存储并重用以前的函数调用结果通常被称为 memoization，这也是一种缓存技术。还有其他几种算法可利用 memoization 来极大地改善性能，这种编程方法通常被称为**动态规划**。

然而，缓存带来的好处并非免费的。实际上，通常以牺牲一些空间为代价来换取应用程序的速度。另外，如果缓存区位于网上，还需要付出传输代价和花费通信时间。你应该进行评估，确定在什么情况下使用缓存可提供便利，以及你愿意以多少空间来换取速度的提升。

鉴于缓存技术很有用，Python 标准库包含了模块 `functools`，让你能够直接使用基于内存的缓存。通过使用装饰器 `functools.lru_cache`，你可轻松地缓存函数的结果。在下面的示例中，我们创建了一个名为 `sum2` 的函数，它打印一条语句并返回两个数字的和。我们运行了这个函数两次。从输出可知，第一次执行函数 `sum2` 时生成了字符串 `"Calculating ..."`，而第二次直接返回结果，没有运行该函数。

```
from functools import lru_cache

@lru_cache()
def sum2(a, b):
    print("Calculating {} + {}".format(a, b))
    return a + b

print(sum2(1, 2))
# 输出：
# Calculating 1 + 2
# 3

print(sum2(1, 2))
# 输出：
# 3
```

装饰器 `lru_cache` 还提供了其他基本功能。要限制缓存区的大小，可使用参数 `max_size`

指定要保留的元素个数；如果希望缓存区大小不受限制，可将这个参数设置为 None。下面是一个使用参数 max_size 的示例。

```
@lru_cache(max_size=16)
def sum2(a, b):
    ...
```



这样，当我们使用不同的参数执行函数 sum2 时，缓存区将逐渐达到最大长度 16，然后再调用这个函数时，缓存区中原来的值将被新计算得到的值覆盖。前缀 lru 就源于这种策略，它表示最近用得最少的（least recently used）。

装饰器 lru_cache 还给被装饰的函数添加了额外的功能。例如，可使用方法 cache_info 来查看缓存的性能，还可使用方法 cache_clear 来清除缓存，如下所示。

```
sum2.cache_info()
# 输出：CacheInfo (hits=0, misses=1, maxsize=128, currsize=1)
sum2.cache_clear()
```

作为示例，我们来看一个可受益于缓存技术的问题——计算斐波纳契数列。我们可这样定义函数 fibonacci 并测量其执行时间：

```
def fibonacci(n):
    if n < 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# 未使用memoization 的版本
%timeit fibonacci(20)
100 loops, best of 3: 5.57 ms per loop
```

执行时间为 5.57 毫秒，这算很长了。这样编写的函数的可伸缩性很糟：由于没有重用之前计算的斐波纳契数列，导致这种算法的时间复杂度大约为 $O(2^N)$。

通过使用缓存技术来存储并重用之前计算得到的斐波纳契数，可改善这种算法的性能。要实现缓存版本，只需对函数 fibonacci 应用装饰器 lru_cache 即可。另外，为设计合适的基准测试程序，需确保每次运行时都实例化新缓存，为此可使用函数 timeit.repeat，如下面的示例所示。

```
import timeit
setup_code = '''
from functools import lru_cache
from __main__ import fibonacci
fibonacci_memoized = lru_cache(maxsize=None)(fibonacci)
'''

results = timeit.repeat('fibonacci_memoized(20)',
                        setup=setup_code,
                        repeat=1000,
```

```
                        number=1)
print("Fibonacci took {:.2f} us".format(min(results)))
# 输出：Fibonacci took 0.01 us
```

虽然我们只是通过添加一个简单的装饰器来修改算法，但现在运行时间远短于 1 毫秒。原因在于由于使用了缓存技术，现在算法的时间复杂度是线性的，而不是指数的。

在应用程序中，可使用装饰器 lru_cache 来实现基于内存的简单缓存。在更复杂的情况下，可使用第三方模块来提供更强大的实现和基于磁盘的缓存。

### joblib

joblib 是一个简单的库，提供了基于磁盘的简单缓存，还有其他功能。这个包的用法与 lru_cache 类似，但结果存储在磁盘中，不会随应用程序的终止而消失。

> 模块 joblib 可在 PyPI 中找到，你可使用命令 pip install joblib 来安装它。

模块 joblib 提供了 Memory 类，你可使用装饰器 Memory.cache 来存储函数的结果。

```
from joblib import Memory
memory = Memory(cachedir='/path/to/cachedir')

@memory.cache
def sum2(a, b):
    return a + b
```

这个装饰器的作用与 lru_cache 类似，但结果将存储在磁盘中——初始化 Memory 时通过参数 cachedir 指定的目录中。另外，缓存的结果不会随应用程序的终止而消失！

使用方法 Memory.cache 还可指定仅在特定参数发生变化时才重新计算，这让被装饰的函数具备清除和分析缓存的基本功能。

由于使用了智能散列算法，joblib 最大的特色在于，能够对操作 NumPy 数组的函数进行高效的 memoization，这在科学和工程应用程序中很有用。

## 2.3　推导和生成器

本节将探索一些使用推导和生成器改善 Python 循环的性能的简单策略。在 Python 中，推导和生成器表达式是经过极度优化的操作，非常适合用来替代显式的 for 循环。使用它们的另一个原因是代码的可读性更强：即便速度不比标准循环高多少，但推导和生成器语法更紧凑，且在大多数情况下更直观。

从下面的示例可知，与函数 sum 结合使用时，列表推导和生成器表达式的速度都比显式循环要快。

```
def loop():
    res = []
    for i in range(100000):
        res.append(i * i)
    return sum(res)

def comprehension():
    return sum([i * i for i in range(100000)])

def generator():
    return sum(i * i for i in range(100000))

%timeit loop()
100 loops, best of 3: 16.1 ms per loop
%timeit comprehension()
100 loops, best of 3: 10.1 ms per loop
%timeit generator()
100 loops, best of 3: 12.4 ms per loop
```

与列表一样，使用字典推导来生成字典时，效率也要高些，代码也更紧凑，如下面的代码所示。

```
def loop():
    res = {}
    for i in range(100000):
        res[i] = i
    return res

def comprehension():
    return {i: i for i in range(100000)}
%timeit loop()
100 loops, best of 3: 13.2 ms per loop
%timeit comprehension()
100 loops, best of 3: 12.8 ms per loop
```

要实现高效的循环（尤其是在内存使用方面），可结合使用迭代器和 filter、map 等函数。例如，来看这样一个问题，即使用列表推导对列表执行一系列操作，再提取最大的值。

```
def map_comprehension(numbers):
    a = [n * 2 for n in numbers]
    b = [n ** 2 for n in a]
    c = [n ** 0.33 for n in b]
    return max(c)
```

这种方法存在的问题是，对于每个列表推导都将分配一个新列表，这增加了内存使用量。我们可使用生成器，而不使用列表推导。生成器是对象，对其进行迭代时，将每次计算一个值并返回结果。

例如，函数 map 接受两个参数——一个函数和一个迭代器，并返回一个生成器，该生成器

将函数应用于集合中的每个元素。这里的重点在于，这种操作是在**迭代期间**进行的，而不是在调用函数 map 时进行的！

我们可使用 map 来重写前面的函数，这将创建中间生成器（而不是列表），从而动态地计算值以节省内存。

```
def map_normal(numbers):
    a = map(lambda n: n * 2, numbers)
    b = map(lambda n: n ** 2, a)
    c = map(lambda n: n ** 0.33, b)
    return max(c)
```

在 IPython 会话中，可使用扩展 memory_profiler 来剖析这两种解决方案的内存使用情况。这个扩展提供了实用工具%memit，可像%timeit 那样帮助我们评估 Python 语句的内存使用情况，如下所示。

```
%load_ext memory_profiler
numbers = range(1000000)
%memit map_comprehension(numbers)
peak memory: 166.33 MiB, increment: 102.54 MiB
%memit map_normal(numbers)
peak memory: 71.04 MiB, increment: 0.00 MiB
```

如你所见，第一个版本占用的内存为 102.54MiB，而第二个版本占用的内存为 0.00MiB！有兴趣的读者可在模块 itertools 中找到其他返回迭代器的函数，这个模块提供了一组设计用来处理常见迭代模式的实用工具。

## 2.4　小结

通过优化算法，可改善应用程序的可伸缩性，使其能够处理更多的数据。本章演示了一些最常见的 Python 数据结构（如列表、双端队列、字典、堆和字典树）的用途和运行时间；介绍了缓存，这是一种以牺牲一些空间（内存或磁盘）为代价，来提高应用程序响应速度的技术；还演示了通过将 for 循环替换为速度更快的结构，如列表推导和生成器表达式，可适度地提高速度。

接下来的两章将介绍如何使用 NumPy 等库来进一步改善性能，以及如何通过 Cython 使用低级语言编写扩展模块。

第 3 章

# 使用 NumPy 和 Pandas 快速执行数组操作



NumPy 是 Python 科学计算的事实标准，它让 Python 支持灵活的多维数组，让数学计算快速而简明。

NumPy 提供了一些常用的数据结构和算法，让你能够使用简明的语法来表示复杂的数学运算。在内部，多维数组 `numpy.ndarray` 是基于 C 语言数组的。这种选择除了可以提高性能，还让 NumPy 代码能够轻松地与既有的 C 和 FORTRAN 例程互操作，从而在使用这些语言编写的遗留代码和 Python 之间搭建桥梁。

本章将介绍如何创建和操作 NumPy 数组，还将探索 NumPy 的广播功能，它让你能够以高效而简明的方式重写复杂的数学表达式。

Pandas 是一个严重依赖于 NumPy 的工具，同时提供了其他致力于数据分析的数据结构和算法。我们将介绍 Pandas 的主要功能及其用法，还将介绍如何使用 Pandas 的数据结构和向量化操作（vectorized operation）来实现高性能。

本章介绍如下主题：

- 创建和操作 NumPy 数组；
- 掌握 NumPy 的广播功能以快速而简明地执行向量化操作；
- 使用 NumPy 改进粒子模拟器；
- 使用 `numexpr` 最大限度地优化性能；
- Pandas 基础知识；
- 使用 Pandas 执行数据库式操作。

## 3.1 NumPy 基础

NumPy 库的核心是其多维数组对象 `numpy.ndarray`。NumPy 数组是由一系列数据类型相

同的元素组成的集合，这种基本限制让 NumPy 能够以特定的方式封装数据，从而能够高性能地执行数学运算。

### 3.1.1　创建数组

要创建 NumPy 数组，可使用函数 numpy.array。这个函数将一个类似于列表的对象（或另一个数组）作为输入，还接受一个可选参数——表示元素数据类型的字符串。通过使用 IPython shell，可交互地测试数组创建代码，如下所示。

```
import numpy as np
a = np.array([0, 1, 2])
```

每个 NumPy 数组都有相关联的数据类型，这可使用属性 dtype 来访问。对于前面的数组 a，其 dtype 为 int64，这表示 64 位整数。

```
a.dtype
# 结果：
# dtype('int64')
```

你可能想将这些整数转换为 float 类型，为此可在初始化数组时传入参数 dtype，也可使用方法 astype 将数组转换为其他数据类型，如下面的代码所示。

```
a = np.array([1, 2, 3], dtype='float32')
a.astype('float32')
# 结果：
# array([ 0.,  1.,  2.], dtype=float32)
```

要创建二维数组（由数组组成的数组），可在初始化时使用嵌套序列，如下所示。

```
a = np.array([[0, 1, 2], [3, 4, 5]])
print(a)
# 输出：
# [[0 1 2]
#  [3 4 5]]
```

以这种方式创建的数组是二维的，但 NumPy 将维度称为轴（axes）。这个数组就像一个包含两行三列的表格。要访问轴，可使用属性 ndarray.shape。

```
a.shape
# 结果：
# (2, 3)
```

你还可以调整数组的形状，条件是各维度的长度的乘积与数组的元素总数相等，即保持总元素个数不变。例如，对于包含 16 个元素的数组，可像下面这样调整其形状：(2, 8)、(4, 4) 或 (2, 2, 4)。要调整数组的形状，可使用方法 ndarray.reshape，也可给重新给元组 ndarray.shape 指定值。下面的代码演示了方法 ndarray.reshape 的用法。

```
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8,
              9, 10, 11, 12, 13, 14, 15])
a.shape
# 输出：
# (16,)

a.reshape(4, 4) # 等价于 a.shape =(4, 4)
# 输出：
# array([[ 0,  1,  2,  3],
#        [ 4,  5,  6,  7],
#        [ 8,  9, 10, 11],
#        [12, 13, 14, 15]])
```

由于这种特征，你可随便添加长度为 1 的维度。对于包含 16 个元素的数组，你可将其形状调整为(16, 1)、(1, 16)、(16, 1, 1)等。在下一节，我们将通过**广播**利用这种功能来执行复杂的操作。

3

NumPy 提供了一些便利函数，可用于创建使用 0 或 1 填充的数组以及没有初始值的数组（在这种情况下，数组的实际值取决于内存状态，因此毫无意义），如下面的代码所示。这些函数将以元组表示的数组形状作为参数，还有一个表示数据类型的可选参数 dtype。

```
np.zeros((3, 3))
np.empty((3, 3))
np.ones((3, 3), dtype='float32')
```

在我们的示例中，我们将使用模块 numpy.random 来生成位于区间(0, 1)内的随机浮点数。numpy.random.rand 将一个表示形状的元组作为参数，并返回一个指定形状的随机数数组。

```
np.random.rand(3, 3)
```

在有些情况下，如果能够初始化一个与其他数组形状相同的数组将会很方便。有鉴于此，NumPy 提供了一些便利函数，如 zeros_like、empty_like 和 ones_like。这些函数的用法如下：

```
np.zeros_like(a)
np.empty_like(a)
np.ones_like(a)
```

### 3.1.2 访问数组

从表面上看，NumPy 数组的接口与 Python 列表类似。对于 NumPy 数组，可使用整数索引来访问其元素，还可使用 for 循环来迭代其元素。

```
A = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
A[0]
# 结果：
# 0

[a for a in A]
```

```
# 结果：
# [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

在 NumPy 中，可在下标运算符`[]`中使用多个用逗号分隔的索引来访问数组元素和子数组。如果有一个`(3,3)`的数组（包含 3 个三元组的数组），访问索引为 0 的元素，得到的将是第一行，如下所示。

```
A = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
A[0]
# 结果：
# array([0, 1, 2])
```

对于特定的行，可再使用一个用逗号分隔的索引来访问其中的元素。例如，要访问第一行的第二个元素，可使用索引`(0, 1)`。需要指出的一个要点是，`A[0, 1]`实际上是`A[(0, 1)]`的简写，换而言之，索引实际上是一个**元组**！下面的代码演示了这两种版本。

```
A[0, 1]
# 结果：
# 1

# 使用元组的等价版本
A[(0, 1)]
```

NumPy 支持在多个维度上对数组执行切片操作。如果在第一维上执行切片操作，将得到一个由三元组组成的集合，如下所示。

```
A[0:2]
# 结果：
# array([[0, 1, 2],
#        [3, 4, 5]])
```

如果再使用 `0:2` 在第二维上对数组执行切片操作，将相当于从前述每个三元组中提取前两个元素，结果是一个形状为`(2, 2)`的数组，如下面的代码所示。

```
A[0:2, 0:2]
# 结果：
# array([[0, 1],
#        [3, 4]])
```

要更新数组中的值，可使用数字索引，也可使用切片，如下面的代码所示。

```
A[0, 1] = 8
A[0:2, 0:2] = [[1, 1], [1, 1]]
```

使用切片语法来访问数组的速度非常快，因为不同于列表，这不会创建数组的副本。用 NumPy 的话说，这将返回原始内存区域的一个**视图**。如果我们获取原始数组的一个切片，并修改其中的一个值，原始数组也将被修改。下面的代码演示了这一点。

```
a= np.array([1, 1, 1, 1])
a_view = a[0:2]
```

```
a_view[0] = 2
print(a)
# 输出：
# [2 1 1 1]
```

修改 NumPy 数组时务必万分小心。由于数据是在视图之间共享的，因此修改视图中的值可能导致难以找出的 bug。为避免副作用，可将标志 `a.flags.writeable` 设置为 `False`，这将避免你无意间修改数组或其视图。

下面再来看一个示例，它演示了在实际工作中如何使用切片语法。我们定义了一个名为 `r_i` 的数组（如下面的代码行所示），它包含 10 个$(x, y)$坐标，因此形状为`(10, 2)`。



```
r_i = np.random.rand(10, 2)
```

> TIP 如果你无法区分轴排列顺序不同的数组，如形状分别为`(10, 2)`和`(2, 10)`的数组，这样想将大有帮助：每当说一个“的”字，都将增加一维。例如，包含 10 个元素，而每个元素的长度都为 2 的数组为`(10, 2)`。相反，包含两个元素，而每个元素的长度都为 10 的数组为`(2, 10)`。

我们要执行的一种典型操作可能是提取每个坐标中的 $x$ 部分。换而言之，你可能想提取索引分别为`(0, 0)`、`(1, 0)`、`(2, 0)`等的元素，结果是一个形状为`(10,)`的数组。在这种情况下，这样想大有帮助：第一个索引是不断**变化**的，而第二个索引是**固定**的（始终为 0）。知道这一点后，我们就可在第一个轴（不断变化的轴）上执行切片操作，并在第二个轴上提取第一个元素（固定的元素），如下面的代码行所示。

```
x_i = r_i[:, 0]
```

相反，下面的表达式保持第一个索引不变，同时不断修改第二个索引，因此返回第一个$(x, y)$坐标。

```
r_0 = r_i[0, :]
```

对最后一个轴执行覆盖所有索引的切片操作时，可显式地指定，也可省略，因此 `r_i[0]`和`r_i[0, :]`等效。

可将由整数或布尔值组成的一个 NumPy 数组作为索引来访问另一个 NumPy 数组，这称为**花式索引**（fancy indexing）。

如果你将一个由整数组成的数组（假设为 `idx`）作为索引来访问另一个数组（假设为 `a`），NumPy 将把这些整数视为索引，并返回一个包含相应值的数组。如果你将 `np.array([0, 2, 3])` 作为索引来访问一个包含 10 个元素的数组，将获得一个形状为`(3,)`的数组，其中包含原始数组中索引分别为 0、2 和 3 的元素。下面的代码演示了这种概念。

```
a = np.array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
idx = np.array([0, 2, 3])
```

```
a[idx]
# 结果：
# array([9, 7, 6])
```

你可使用花式索引来访问多维数组，方法是为每维都指定一个数组。例如，如果你要提取索引分别为`(0, 2)`和`(1, 3)`的元素，就必须将所有针对第一个轴的索引都放在一个数组中，并将所有针对第二个轴的索引都放在另一个数组中，如下面的代码所示。

```
a = np.array([[0, 1, 2], [3, 4, 5],
              [6, 7, 8], [9, 10, 11]])
idx1 = np.array([0, 1])
idx2 = np.array([2, 3])
a[idx1, idx2]
```

你还可将普通列表用作索引数组，但元组不可以。例如，下面的两条语句等价。

```
a[np.array([0, 1])] # 与下面的语句等价
a[[0, 1]]
```

然而，如果你将元组用作索引，NumPy 将把它视为针对多个维度的索引。

```
a[(0, 1)] # 与下面的语句等价
a[0, 1]
```

索引数组并非必须是一维的，我们可以以任何形状提取原始数组中的元素。例如，可以从原始数组中提取元素，组成一个`(2, 2)`的数组，如下所示。

```
idx1 = [[0, 1], [3, 2]]
idx2 = [[0, 2], [1, 1]]
a[idx1, idx2]
# 输出：
# array([[ 0, 5],
#        [10, 7]])
```

可结合使用数组切片和花式索引功能，这在需要交换坐标数组中的 $x$ 和 $y$ 列时很有用。在下面的代码中，第一个索引是一个切片，覆盖了所有的元素。对于每个元素，我们先提取位置 1 的值（$y$ 坐标），再提取位置 0 的值（$x$ 坐标）。

```
r_i = np.random(10, 2)
r_i[:, [0, 1]] = r_i[:, [1, 0]]
```

索引数组为布尔类型时，规则稍有不同。在这种情况下，布尔数组犹如**掩码**：提取每个与`True`对应的元素，并将其加入到输出数组中，如下面的代码所示。

```
a = np.array([0, 1, 2, 3, 4, 5])
mask = np.array([True, False, True, False, False, False])
a[mask]
# 输出：
# array([0, 2])
```

涉及多个维度时，这些规则也适用。另外，如果索引数组与原始数组的形状相同，将选择与 True 对应的元素，并将其加入到输出数组中。

在 NumPy 中，索引操作的速度相当快。虽然如此，在速度至关重要时，可使用速度更快的函数 numpy.take 和 numpy.compress 来进一步提高性能。函数 numpy.take 的第一个参数是要操作的数组，第二个参数是由要提取的元素的索引组成的列表。最后一个参数为 axis；如果没有指定，索引将应用于扁平化后的数组（flattened array），否则将应用于指定的轴。

```
r_i = np.random(100, 2)
idx = np.arange(50) # 整数 0～50

%timeit np.take(r_i, idx, axis=0)
1000000 loops, best of 3: 962 ns per loop

%timeit r_i[idx]
100000 loops, best of 3: 3.09 us per loop
```

与此类似的是 numpy.compress，其速度更快，用于布尔数组，但工作原理没什么不同。下面演示了 numpy.compress 的用法：

```
In [51]: idx = np.ones(100, dtype='bool') # 所有元素的值都为 True
In [52]: %timeit np.compress(idx, r_i, axis=0)
1000000 loops, best of 3: 1.65 us per loop
In [53]: %timeit r_i[idx]
100000 loops, best of 3: 5.47 us per loop
```

### 3.1.3 广播

NumPy 的真正威力在于其快速的数学运算。NumPy 使用经过优化的 C 语言代码来执行基于元素的计算，从而避开了 Python 解释器。**广播**是一组巧妙的规则，使得能够对形状类似（但不完全相同）的数组快速地执行数组计算。

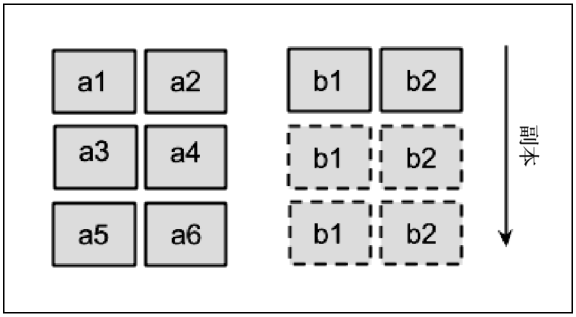
每当你对两个数组执行算术运算（如乘积）时，如果这两个操作数的形状相同，将以逐元素的方式执行运算。例如，将两个形状为(2, 2)的数组相乘时，将把对应的元素对相乘，得到一个(2, 2)的数组，如下面的代码所示。

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
A * B
# 输出：
# array([[ 5, 12],
#        [21, 32]])
```

如果两个操作数的形状不匹配，NumPy 将尝试使用广播规则让它们匹配。如果其中一个操作数为**标量**（如数字），将把它应用于数组的每个元素，如下面的代码所示。

```
A * 2
# 输出:
# array([[2, 4],
#        [6, 8]])
```

如果两个操作数都是数组，NumPy 将尝试从最后一个轴开始让它们的形状匹配。例如，如果要合并两个形状分别为(3, 2)和(2,)的数组，将把第二个数组重复 3 次，生成一个(3, 2)的数组。换而言之，将沿一个维度**广播**这个数组，使其形状与另一个操作数匹配，如下图所示。




如果形状不匹配，例如合并两个形状分别为(3, 2)和(2, 2)的数组时，NumPy 将引发异常。

如果一个轴的长度为 1，将沿这个轴重复数组，直到形状匹配。为演示这一点，假设有一个形状如下的数组：

```
5, 10, 2
```

现在假设我们要在这个数组中广播一个形状为(5, 1, 2)的数组，这将在第二个轴上重复第二个数组 10 次，如下所示。

```
5, 10, 2
5,  1, 2 →重复
- - - -
5, 10, 2
```

前面说过，可通过添加长度为 1 的轴来调整数组的形状。通过将常量 `numpy.newaxis` 用作索引，可引入一个维度。例如，如果有一个形状为(5, 2)的数组，要将它与一个形状为(5, 10, 2)的数组合并，可在中间添加一个轴，以得到一个形状为(5, 1, 2)的兼容数组，如下面的代码所示。

```
A = np.random.rand(5, 10, 2)
B = np.random.rand(5, 2)
A * B[:, np.newaxis, :]
```

例如，可利用这一点来操作两个数组的所有可能组合。一个这样的用途是**外积**。假设有如下两个数组：

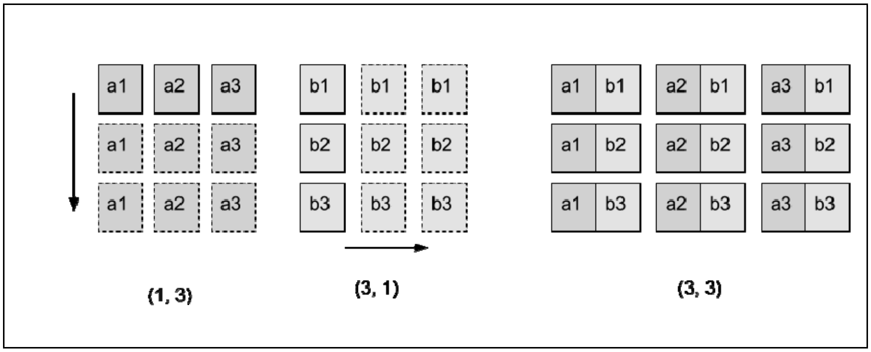
```
a = [a1, a2, a3]
b = [b1, b2, b3]
```

外积是一个矩阵，包含两个数组中元素的各种组合的乘积，如下面的代码所示。

```
a x b = a1*b1, a1*b2, a1*b3
        a2*b1, a2*b2, a2*b3
        a3*b1, a3*b2, a3*b3
```

为使用 NumPy 计算外积，我们在一个维度上重复元素`[a1, a2, a3]`，并在另一个维度上重复元素`[b1, b2, b3]`，再计算对应元素的乘积，如下图所示。






用代码表示时，我们的策略是将数组 `a` 从形状`(3,)`转换为形状`(3, 1)`，并将数组 `b` 从形状`(3,)`转换为形状`(1, 3)`。这样，将在两个维度上分别广播这两个数组，并将对应的元素相乘，如下面的代码所示。

```
AB = a[:, np.newaxis] * b[np.newaxis, :]
```

由于避免了 Python 循环，这种操作的速度极快且效果极佳，在处理大量元素时，其速度可与纯粹的 C 或 FORTRAN 代码媲美。

### 3.1.4 数学运算

NumPy 默认支持使用广播技术来执行最常见的数学运算，这包括简单的代数运算、三角运算、取整和逻辑运算。例如，要对数组中每个元素取平方根，可使用 `numpy.sqrt`，如下面的代码所示。

```
np.sqrt(np.array([4, 9, 16]))
# 结果:
# array([2., 3., 4.])
```

在根据条件筛选元素时，比较运算符很有用。假设有一个由 0~1 的随机数组成的数组，而你要提取其中所有大于 0.5 的数字，为此可对这个数组使用运算符>，这将得到一个布尔数组，如

下所示。

```
a = np.random.rand(5, 3)
a > 0.3
# 结果:
# array([[ True, False,  True],
#        [ True,  True,  True],
#        [False,  True,  True],
#        [ True,  True, False],
#        [ True,  True, False]], dtype=bool)
```

然后，可将这个布尔数组作为索引来获取大于 0.5 的元素。

```
a[a > 0.5]
print(a[a>0.5])
# 输出:
# [ 0.9755  0.5977  0.8287  0.6214  0.5669  0.9553  0.5894
0.7196  0.9200  0.5781  0.8281 ]
```

NumPy 还实现了方法 `ndarray.sum`，它计算特定轴上所有元素的和。例如，如果有一个形状为`(5, 3)`的数组，可使用方法 `ndarray.sum` 来计算第一个轴上所有元素、第二个轴上所有元素或该数组中所有元素的和，如下面的代码所示。

```
a = np.random.rand(5, 3)
a.sum(axis=0)
# 结果:
# array([ 2.7454, 2.5517, 2.0303])

a.sum(axis=1)
# 结果:
# array([ 1.7498, 1.2491, 1.8151, 1.9320, 0.5814])

a.sum() # 没有指定参数时，将作用于扁平化后的数组
# 结果:
# 7.3275
```

请注意，通过在特定轴上求和，将消除这个轴。从前面的示例可知，在轴 0 上求和得到的是一个形状为`(3,)`的数组，而在轴 1 上求和得到的是一个形状为`(5,)`的数组。

### 3.1.5　计算范数

为复习前面介绍的基本概念，我们来计算一组坐标的**范数**。对于二维向量，范数的定义如下：

```
norm = sqrt(x**2 + y**2)
```

给定一个包含 10 个$(x, y)$坐标的数组，我们要计算每个坐标的范数。为计算范数，可采取如下步骤。

(1) 计算坐标值的平方，得到一个元素值为`(x**2, y**2)`的数组。

(2) 使用 numpy.sum 在最后一个轴上将元素相加。
(3) 使用 numpy.sqrt 计算每个元素的平方根。

最终的表达式可压缩成一行代码：

```
r_i = np.random.rand(10, 2)
norm = np.sqrt((r_i ** 2).sum(axis=1))
print(norm)
# 输出：
# [ 0.7314  0.9050  0.5063  0.2553  0.0778  0.9143  1.3245
0.9486  1.010  1.0212]
```



## 3.2 使用 NumPy 重写粒子模拟器

本节将优化粒子模拟器——使用 NumPy 重写其中一些部分。从第 1 章所做的剖析可知，在这个程序中，最慢的部分是方法 ParticleSimulator.evolve 中的如下循环。

```
for i in range(nsteps):
for p in self.particles:

  norm = (p.x**2 + p.y**2)**0.5
  v_x = (-p.y)/norm
  v_y = p.x/norm

  d_x = timestep * p.ang_vel * v_x
  d_y = timestep * p.ang_vel * v_y

  p.x += d_x
  p.y += d_y
```

你可能注意到了，这个循环体只处理当前粒子。如果我们有一个包含粒子位置和角速度的数组，就可使用广播操作来重写这个循环。但这个循环的每个迭代都依赖于前一个迭代，因此无法采用这种方式进行并行化。

有鉴于此，一种自然而然的选择是，将所有位置坐标都存储在一个形状为(nparticles, 2)的数组中，并将角速度存储在一个形状为(nparticles,)的数组中，其中 nparticles 是粒子总数。我们将这两个数组分别命名为 r_i 和 ang_vel_i：

```
r_i = np.array([[p.x, p.y] for p in self.particles])
ang_vel_i = np.array([p.ang_vel for p in self.particles])
```

速度的方向垂直于向量$(x, y)$，其定义如下：

```
v_x = -y / norm
v_y = x / norm
```

可使用 3.1.5 节演示的策略来计算范数：

```
norm_i = ((r_i ** 2).sum(axis=1))**0.5
```

为计算组分$(-y, x)$，首先需要将数组 `r_i` 的 $x$ 和 $y$ 列交换，再将第一列乘以–1，如下面的代码所示。

```
v_i = r_i[:, [1, 0]] / norm_i
v_i[:, 0] *= -1
```

为计算位移，需要计算 `v_i`、`ang_vel_i` 和 `timestep` 的乘积。由于 `ang_vel_i` 的形状为 `(nparticles,)`，因此需要给它添加一个轴，以便将其与形状为 `(nparticles, 2)` 的 `v_i` 相乘。为此，我们使用 `numpy.newaxis`，如下所示。

```
d_i = timestep * ang_vel_i[:, np.newaxis] * v_i
r_i += d_i
```

在循环外部，我们必须使用新的 $x$ 和 $y$ 坐标更新粒子实例。

```
for i, p in enumerate(self.particles):
  p.x, p.y = r_i[i]
```

总之，我们将实现一个名为 `ParticleSimulator.evolve_numpy` 的方法，并使用基准测试将其同纯粹的 Python 版本（更名为 `ParticleSimulator.evolve_python`）进行比较。

```
def evolve_numpy(self, dt):
  timestep = 0.00001
  nsteps = int(dt/timestep)

  r_i = np.array([[p.x, p.y] for p in self.particles])
  ang_vel_i = np.array([p.ang_vel for p in self.particles])

  for i in range(nsteps):

    norm_i = np.sqrt((r_i ** 2).sum(axis=1))
    v_i = r_i[:, [1, 0]]
    v_i[:, 0] *= -1
    v_i /= norm_i[:, np.newaxis]
    d_i = timestep * ang_vel_i[:, np.newaxis] * v_i
    r_i += d_i

    for i, p in enumerate(self.particles):
      p.x, p.y = r_i[i]
```

我们还将修改基准测试程序，以便方便地调整粒子数和模拟方法，如下所示。

```
def benchmark(npart=100, method='python'):
  particles = [Particle(uniform(-1.0, 1.0),
                        uniform(-1.0, 1.0),
                        uniform(-1.0, 1.0))
                        for i in range(npart)]

  simulator = ParticleSimulator(particles)

  if method=='python':
    simulator.evolve_python(0.1)
```

```
  elif method == 'numpy':
    simulator.evolve_numpy(0.1)
```

下面在 IPython 会话中运行基准测试程序。

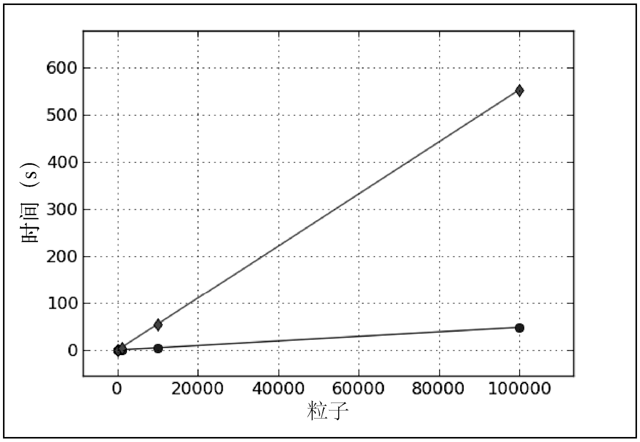
```
from simul import benchmark
%timeit benchmark(100, 'python')
1 loops, best of 3: 614 ms per loop
%timeit benchmark(100, 'numpy')
1 loops, best of 3: 415 ms per loop
```

速度有一定的提升，幅度看起来虽然不是很大，但 NumPy 在处理大型数组方面的威力展示出来了。如果我们增加粒子数，将发现性能提升更为明显。



```
%timeit benchmark(1000, 'python')
1 loops, best of 3: 6.13 s per loop
%timeit benchmark(1000, 'numpy')
1 loops, best of 3: 852 ms per loop
```

下图是根据使用不同的粒子数运行基准测试程序得到的结果绘制而成的。




该图表明，这两种实现的运行时间都与粒子数成正比，但纯粹的 Python 版本的运行时间的增速比 NumPy 版本大得多。粒子数越多，NumPy 的优势越大。一般而言，使用 NumPy 时，应尽量将数据放在大型数组中，并使用广播功能将计算编组。

## 3.3　使用 numexpr 最大限度地提高性能

处理复杂的表达式时，NumPy 将中间结果存储在内存中。David M. Cooke 编写了一个名为 `numexpr` 的包，能够动态地优化并编译数组表达式。这个包还能够优化 CPU 缓存使用量，并利用多个处理器。

这个包基于单个函数 numexpr.evaluate，使用起来通常比较简单。这个函数将一个包含数组表达式的字符串作为第一个参数，其语法与 NumPy 大致相同。例如，可像下面这样来计算简单表达式 a + b * c。

```
a = np.random.rand(10000)
b = np.random.rand(10000)
c = np.random.rand(10000)
d = ne.evaluate('a + b * c')
```

几乎在任何情况下，使用 numexpr 包都可改善性能，但仅当用于处理大型数组时，性能提升才会特别大。一个涉及大型数组的应用程序是计算**距离矩阵**。在粒子系统中，距离矩阵包含任何两个粒子之间的距离。要计算这个矩阵，首先需要计算将任何两个粒子(i,j)连接起来的向量，如下所示。

```
x_ij = x_j - x_i
y_ij = y_j - y_i.
```

接下来，计算这个向量的长度（即范数），如下面的代码所示。

```
d_ij = sqrt(x_ij**2 + y_ij**2)
```

在 NumPy 中，可使用广播规则来编写这样的代码（这种运算类似于外积）:

```
r = np.random.rand(10000, 2)
r_i = r[:, np.newaxis]
r_j = r[np.newaxis, :]
d_ij = r_j - r_i
```

然后，在最后一个轴上计算范数，如下面的代码所示。

```
d_ij = np.sqrt((d_ij ** 2).sum(axis=2))
```

使用 numexpr 语法重写这个表达式很容易。numexpr 包不支持在数组表达式中使用切片，因此我们首先需要在操作数中添加额外的维度以支持广播，如下所示。

```
r = np.random(10000, 2)
r_i = r[:, np.newaxis]
r_j = r[np.newaxis, :]
```

应将尽可能多的操作放在一个表达式中，这样才能实现明显的优化。

大多数 NumPy 数学函数在 numexpr 包中都有，但有一个限制，那就是归约操作（消除一个轴的操作，如 sum）必须最后执行。因此，我们必须先计算总和，再离开 numexpr，然后使用另一个表达式来计算平方根。

```
d_ij = ne.evaluate('sum((r_j - r_i)**2, 2)')
d_ij = ne.evaluate('sqrt(d_ij)')
```

numexpr 编译器不存储中间结果，以避免不必要的内存分配。它还尽可能将运算分给多个处

理器去执行。在文件 distance_matrix.py 中，有两个函数实现了这两个版本：distance_matrix_numpy 和 distance_matrix_numexpr。

```
from distance_matrix import (distance_matrix_numpy,
                             distance_matrix_numexpr)
%timeit distance_matrix_numpy(10000)
1 loops, best of 3: 3.56 s per loop
%timeit distance_matrix_numexpr(10000)
1 loops, best of 3: 858 ms per loop
```

通过对表达式进行转换，以使用 numexpr，就让性能提高到了使用标准 NumPy 的 4.5 倍。每当你需要优化涉及大型数组和复杂运算的 NumPy 表达式时，都可使用 numexpr 包，而且只需对代码做很少的修改。



## 3.4 Pandas

Pandas 是一个设计用于以无缝而高效的方式分析数据集的库，最初是由 Wes McKinney 开发的。最近几年，这个功能强大的库风生水起，被 Python 社区广泛使用。本节将简要地介绍这个库的主要概念及其提供的主要工具，在很多 NumPy 向量化操作和广播技术无能为力的情况下，都可使用它来提升性能。

### 3.4.1 Pandas 基础

NumPy 的主要目标是处理数组，而 Pandas 的主要数据结构为 pandas.Series、pandas.DataFrame 和 pandas.Panel。在本章余下的篇幅中，我们将把 pandas 简称为 pd。

pd.Series 对象和 np.array 的主要不同在于，pd.Series 对象将每个数组元素都关联到一个键。下面通过一个示例来看看其工作原理。

假设我们要测试一种新推出的降压药，对于每位患者，我们都要记录他服用这种新药后血压是否得到了改善。为对这种信息进行编码，可将每个测试对象的 ID（用一个整数表示）关联到 True（如果这种新药有效）或 False（如果无效）。

我们可创建一个 pd.Series 对象，将一个包含键（患者）的数组关联到一个表示效果的数组。可通过参数 index 将键数组传递给构造函数 Series，如下面的代码所示。

```
import pandas as pd
patients = [0, 1, 2, 3]
effective = [True, True, False, False]

effective_series = pd.Series(effective, index=patients)
```

从技术上说，要将一组整数（0~*N*）关联到一组值，也可使用 np.array 来实现，因为在这种情况下，键就是元素在数组中的位置。在 Pandas 中，键不仅可以是整数，还可以是字符串、

浮点数和可散列的 Python 对象。例如，很容易将 ID 变成字符串，如下面的代码所示。

```
patients = ["a", "b", "c", "d"]
effective = [True, True, False, False]

effective_series = pd.Series(effective, index=patients)
```

有趣的是，可将 NumPy 数组视为类似于 Python 列表的连续值集合，可将 Pandas 对象 pd.Series 视为一种将键映射到值的结构，就像 Python 字典。

如果要存储每位患者原来的血压和服药后的血压，该如何做呢？在 Pandas 中，可使用 pd.DataFrame 对象将多项数据关联到同一个键。

要创建 pd.DataFrame，方法与创建 pd.Series 对象类似：传递一个由列构成的字典和一个索引。下面的示例演示了如何创建一个包含 4 列的 pd.DataFrame，这 4 列分别表示服药前后的高压和低压。

```
patients = ["a", "b", "c", "d"]

columns = {
  "sys_initial": [120, 126, 130, 115],
  "dia_initial": [75, 85, 90, 87],
  "sys_final": [115, 123, 130, 118],
  "dia_final": [70, 82, 92, 87]
}

df = pd.DataFrame(columns, index=patients)
```

你可将 pd.DataFrame 视为一个 pd.Series 集合。事实上，可直接使用由 pd.Series 实例组成的字典来创建 pd.DataFrame。

```
columns = {
  "sys_initial": pd.Series([120, 126, 130, 115], index=patients),
  "dia_initial": pd.Series([75, 85, 90, 87], index=patients),
  "sys_final": pd.Series([115, 123, 130, 118], index=patients),
  "dia_final": pd.Series([70, 82, 92, 87], index=patients)
}
df = pd.DataFrame(columns)
```

要查看 pd.DataFrame 和 pd.Series 对象的内容，可分别使用方法 pd.Series.head 和 pd.DataFrame.head，它们将打印数据集的开头几行。

```
effective_series.head()
# 输出：
# a True
# b True
# c False
# d False
# dtype: bool
```

```
df.head()
# 输出:
#    dia_final dia_initial sys_final sys_initial
# a         70          75       115         120
# b         82          85       123         126
# c         92          90       130         130
# d         87          87       118         115
```

pd.DataFrame 可用于存储由 pd.Series 组成的集合，同样，可使用 pd.Panel 来存储由 pd.DataFrames 组成的集合。这里不会介绍 pd.Panel 的用法，因为它不像 pd.Series 和 pd.DataFrame 使用得那么频繁。有关 pd.Panel 的详细信息，请参阅相关的文档。



**访问 Series 和 DataFrame 对象的内容**

要获取 pd.Series 中与指定键相关联的数据，可使用属性 pd.Series.loc 并指定索引。

```
effective_series.loc["a"]
# 结果:
# True
```

也可使用属性 pd.Series.iloc，根据元素在底层数组中的**位置**来访问它。

```
effective_series.iloc[0]
# 结果:
# True
```

还可使用属性 pd.Series.ix 以混合的方式访问元素。在这种情况下，如果指定的值不是整数，将把它视为键来提取相应的元素，否则将把它视为位置来提取相应的元素。直接访问 pd.Series 时，情况与此类似。下面的示例演示了这些概念。

```
effective_series.ix["a"] # 根据键访问
effective_series.ix[0]   # 根据位置访问

# 等价于
effective_series["a"] # 根据键访问
effective_series[0]   # 根据位置访问
```

请注意，如果索引为整数，这个方法将退化为只支持键的方法（就像 loc 一样）。在这种情况下，要根据位置访问元素，只能使用方法 iloc。

访问 pd.DataFrame 的方式与此类似。例如，可使用 pd.DataFrame.loc 根据键来提取相应的行，可使用 pd.DataFrame.iloc 根据位置来提取相应的行。

```
df.loc["a"]
df.iloc[0]
# 结果:
# dia_final 70
# dia_initial 75
# sys_final 115
# sys_initial 120
# Name: a, dtype: int64
```

这里的一个重点是，返回的是一个 pd.Series，其中每列都是一个新键。要获取特定的行和列，可使用下面的代码，其中属性 loc 根据键来确定行和列，而 iloc 根据整数来确定行和列。

```
df.loc["a", "sys_initial"] # 等价于
df.loc["a"].loc["sys_initial"]

df.iloc[0, 1] # 等价于
df.iloc[0].iloc[1]
```

访问 pd.DataFrame 时，可使用属性 ix 以混合的方式指定元素。例如，要获取第 0 行的 sys_initial 列，可像下面这样做：

```
df.ix[0, "sys_initial"]
```

要根据名称获取 pd.DataFrame 中特定的列，可使用常规索引，也可使用属性。要根据位置来获取特定的列，可使用 iloc，也可先使用属性 pd.DataFrame.column 来获取该列的名称。

```
# 根据名称获取列
df["sys_initial"] # 等价于
df.sys_initial

# 根据位置获取列
df[df.columns[2]] # 等价于
df.iloc[:, 2]
```

这些方法还支持类似于 NumPy 中的复杂索引，如布尔值、列表和整型数组。

现在该谈谈性能方面了。Pandas 中的索引与字典的索引有些不同。例如，在字典中，每个键都必须是独一无二的，而 Pandas 索引可包含重复的元素。然而，这种灵活性是要付出代价的：在索引不是独一无二的情况下，元素的访问性能将急剧降低，其时间复杂度为 $O(N)$（就像线性查找），而不像字典那样为 $O(1)$。

为减轻这种影响，一种方法是对索引排序，这样 Pandas 就可使用计算复杂度为 $O(\log(N))$的二分查找法，其性能比线性查找高得多。要对索引进行排序，可使用函数 pd.Series.sort_index，如下面的代码所示（这也适用于 pd.DataFrame）。

```
# 创建一个包含重复索引的 Series
index = list(range(1000)) + list(range(1000))

# 访问常规 Series 的时间复杂度为 O(N)
series = pd.Series(range(2000), index=index)

# 通过排序，可改善查找操作的时间复杂度，使其为 O(log(N))
series.sort_index(inplace=True)
```

下表总结了不同版本的时间复杂度。

| 索引类型 | *N*=10 000 | *N*=20 000 | *N*=30 000 | 时　间 |
|---|---|---|---|---|
| 独一无二 | 12.30 | 12.58 | 13.30 | $O(1)$ |
| 不独一无二 | 494.95 | 814.10 | 1129.95 | $O(N)$ |
| 不独一无二（经过排序） | 145.93 | 145.81 | 145.66 | $O(\log(N))$ |

## 3.4.2 使用 Pandas 执行数据库式操作

你可能注意到了，表格式数据类似于数据库中存储的数据。数据库通常是根据主键访问的，其中各列的数据类型可以不同，就像 `pd.DataFrame` 一样。

在 Pandas 中，索引操作的效率很高，因此可执行数据库式操作，如计数、连接、分组和聚合。



### 1. 映射

与 NumPy 一样，Pandas 支持逐元素操作（毕竟 `pd.Series` 是使用 `np.array` 来存储数据的）。例如，可轻松地对 `pd.Series` 和 `pd.DataFrame` 进行变换。

```
np.log(df.sys_initial) # 计算Series的对数
df.sys_initial ** 2    # 计算Series的平方
np.log(df)             # 计算DataFrame的对数
df ** 2                # 计算DataFrame的平方
```



还可像 NumPy 中那样，对两个 `pd.Series` 执行逐元素运算，但一个重要的不同是，将根据键而不是位置来匹配操作数，如果索引不匹配，结果将为 `NaN`。下面的示例演示了这些情况。

```
# 索引匹配
a = pd.Series([1, 2, 3], index=["a", "b", "c"])
b = pd.Series([4, 5, 6], index=["a", "b", "c"])
a + b
# 结果：
# a 5
# b 7
# c 9
# dtype: int64

# 索引不匹配
b = pd.Series([4, 5, 6], index=["a", "b", "d"])
# 结果：
# a 5.0
# b 7.0
# c NaN
# d NaN
# dtype: float64
```

为增加灵活性，Pandas 暴露了方法 `map`、`apply` 和 `applymap`，你可使用它们来执行特定的变换。

方法 pd.Series.map 可用来对每个值执行指定的函数，它返回一个包含结果的 pd.Series。下面的示例演示了如何对 pd.Series 的每个元素执行函数 superstar。

```
a = pd.Series([1, 2, 3], index=["a", "b", "c"])
def superstar(x):
    return '*' + str(x) + '*'
a.map(superstar)

# 结果:
# a *1*
# b *2*
# c *3*
# dtype: object
```

函数 pd.DataFrame.applymap 与 pd.Series.map 的作用相同，但用于 DataFrame。

```
df.applymap(superstar)
# 结果:
#    dia_final  dia_initial  sys_final  sys_initial
# a       *70*         *75*      *115*        *120*
# b       *82*         *85*      *123*        *126*
# c       *92*         *90*      *130*        *130*
# d       *87*         *87*      *118*        *115*
```

最后，函数 pd.DataFrame.apply 对每列或每行（而不是每个元素）执行传入的函数。要指定对每列还是每行执行指定的函数，可使用参数 axis：其值为 0（默认值）时对每列执行指定的函数，为 1 时对每行执行指定的函数。另外，请注意，这个函数的返回值是一个 pd.Series。

```
df.apply(superstar, axis=0)
# 结果:
# dia_final *a 70nb 82nc 92nd 87nName: dia...
# dia_initial *a 75nb 85nc 90nd 87nName: dia...
# sys_final *a 115nb 123nc 130nd 118nName:...
# sys_initial *a 120nb 126nc 130nd 115nName:...
# dtype: object

df.apply(superstar, axis=1)
# 结果:
# a *dia_final 70ndia_initial 75nsys_f...
# b *dia_final 82ndia_initial 85nsys_f...
# c *dia_final 92ndia_initial 90nsys_f...
# d *dia_final 87ndia_initial 87nsys_f...
# dtype: object
```

Pandas 还通过便利方法 eval 支持高效的 numexpr 式表达式。例如，如果要计算服药前后的血压差，可以字符串的方式编写相应的表达式，如下面的代码所示。

```
df.eval("sys_final - sys_initial")
# 结果:
# a -5
# b -3
```

```
# c 0
# d 3
# dtype: int64
```

还可在传递给 pd.DataFrame.eval 的表达式中使用赋值运算符来创建新列。请注意，如果将参数 inplace 设置成了 True，将直接在原始 pd.DataFrame 上操作，否则这个函数将返回一个新的 DataFrame。下面的示例计算 sys_final 和 sys_initial 的差，并将结果存储在 sys_delta 列中。

```
df.eval("sys_delta = sys_final - sys_initial", inplace=False)
# 结果：
#     dia_final   dia_initial   sys_final   sys_initial   sys_delta
# a          70            75         115           120          -5
# b          82            85         123           126          -3
# c          92            90         130           130           0
# d          87            87         118           115           3
```

3

### 2. 分组、聚合和变换

Pandas 最令人称道的特征之一是，能够以简洁的方式表示对数据进行分组、变换和聚合的数据分析管道。为演示这个概念，我们扩展前面的数据集，在其中添加两位没有服用新药的患者（这通常称为**对照组**）。另外，再增加一列，用于记录患者是否服用了新药。

```
patients = ["a", "b", "c", "d", "e", "f"]

columns = {
  "sys_initial": [120, 126, 130, 115, 150, 117],
  "dia_initial": [75, 85, 90, 87, 90, 74],
  "sys_final": [115, 123, 130, 118, 130, 121],
  "dia_final": [70, 82, 92, 87, 85, 74],
  "drug_admst": [True, True, True, True, False, False]
}

df = pd.DataFrame(columns, index=patients)
```

此时，我们可能想知道两组患者的血压变化情况有何不同。为此，可使用函数 pd.DataFrame.groupby 根据 drug_amst 列对患者进行分组。这个函数返回一个 DataFrameGroupBy 对象，可通过迭代它来获得一系列 pd.DataFrame，它们分别对应于 drug_admst 列不同的值。

```
    df.groupby('drug_admst')
    for value, group in df.groupby('drug_admst'):
        print("Value: {}".format(value))
        print("Group DataFrame:")
        print(group)
# 输出：
# Value: False
# Group DataFrame:
#    dia_final   dia_initial   drug_admst   sys_final   sys_initial
# e         85            90        False         130           150
# f         74            74        False         121           117
```

```
# Value: True
# Group DataFrame:
#    dia_final   dia_initial   drug_admst   sys_final   sys_initial
# a         70            75         True         115           120
# b         82            85         True         123           126
# c         92            90         True         130           130
# d         87            87         True         118           115
```

几乎在任何情况下，都无须迭代 DataFrameGroupBy 对象，因为可直接计算与分组相关的属性，这都是方法串接的功劳。例如，我们可能想计算每个分组的平均值、最大值或标准偏差。以某种方式对数据进行汇总的运算都称为聚合，可使用方法 agg 来完成。这个方法返回一个将分组变量与聚合结果关联起来的 pd.DataFrame，如下面的代码所示。

```
df.groupby('drug_admst').agg(np.mean)
#              dia_final   dia_initial   sys_final   sys_initial
# drug_admst
# False            79.50         82.00       125.5        133.50
# True             82.75         84.25       121.5        122.75
```

也可对并非表示汇总的 DataFrame 分组进行处理，一个这样的常见操作是补全缺失的值。这些中间步骤称为**变换**。

下面通过一个示例来演示这个概念。假设数据集中缺失了一些值，而我们想将它们设置为同一分组中其他值的平均值，为此可使用如下变换。

```
df.loc['a','sys_initial'] = None
df.groupby('drug_admst').transform(lambda df: df.fillna(df.mean()))
#     dia_final    dia_initial   sys_final   sys_initial
# a          70             75         115    123.666667
# b          82             85         123    126.000000
# c          92             90         130    130.000000
# d          87             87         118    115.000000
# e          85             90         130    150.000000
# f          74             74         121    117.000000
```

### 3. 连接

为聚合分散在不同表中的数据，连接很有用。假设我们要在前述数据集中包含患者接受治疗的医院的位置。可使用标签 H1、H2 和 H3 来表示患者是在哪家医院接受的治疗，并将医院的地址和名称存储在 hospital 表中。

```
hospitals = pd.DataFrame(
  { "name" : ["City 1", "City 2", "City 3"],
    "address" : ["Address 1", "Address 2", "Address 3"],
    "city": ["City 1", "City 2", "City 3"] },
  index=["H1", "H2", "H3"])

hospital_id = ["H1", "H2", "H2", "H3", "H3", "H3"]
df['hospital_id'] = hospital_id
```

现在我们要确定每位患者是在哪座城市接受的治疗，为此需要将 hospital_id 列中的键**映射**到存储在 hospitals 表中的城市。

在 Python 中，这可使用字典来实现。

```
hospital_dict = {
 "H1": ("City 1", "Name 1", "Address 1"),
 "H2": ("City 2", "Name 2", "Address 2"),
 "H3": ("City 3", "Name 3", "Address 3")
}
cities = [hospital_dict[key][0]
           for key in hospital_id]
```



这种算法的效率很高，其时间复杂度为 $O(N)$，其中 $N$ 是 hospital_id 的长度。Pandas 让你能够使用简单索引执行上面的操作，其优点在于连接将使用经过高度优化的 Cython 和高效的散列算法来执行。对于前述简单的 Python 表达式，可轻松地将其转换为 Pandas 代码，如下所示。

```
cities = hospitals.loc[hospital_id, "city"]
```

要执行更复杂的连接，可使用方法 pd.DataFrame.join，它将生成一个新的 pd.DataFrame，将患者关联到其接受治疗的医院的信息。

```
result = df.join(hospitals, on='hospital_id')
result.columns
# 结果：
# Index(['dia_final', 'dia_initial', 'drug_admst',
# 'sys_final', 'sys_initial',
# 'hospital_id', 'address', 'city', 'name'],
# dtype='object')
```

## 3.5 小结

本章介绍了如何操作 NumPy 数组，以及如何使用数组广播技术编写快速的数学表达式。利用这些知识，你可编写更简洁、表达力更丰富的代码，同时极大地改善性能。本章还介绍了 numexpr 库，通过使用它，你只需做少量的工作就可进一步提高 NumPy 计算的速度。

Pandas 实现了一些对分析大型数据集很有帮助的高效的数据结构。具体地说，Pandas 擅长处理将非整数键作为索引的数据，它还提供了速度极快的散列算法。

处理大型同质输入时，NumPy 和 Pandas 很好用，但当表达式非常复杂、无法使用这些库提供的工具来表示其中的操作时，它们就不适用了。在这种情况下，可利用 Python 乃胶水语言的特点，使用 Cython 包来与 C 语言交互。

# 第 4 章 使用 Cython 获得 C 语言性能

Cython 是一种扩展 Python 的语言，这是通过支持给函数、变量和类声明类型来实现的。这些类型声明让 Cython 能够将 Python 脚本编译成高效的 C 语言代码。Cython 还可充当 Python 和 C 语言之间的桥梁，因为它提供了易于使用的结构，让你能够编写到外部 C 和 C++例程的接口。

本章介绍如下主题：

- Cython 的基本语法；
- 如何编译 Cython 程序；
- 如何使用**静态类型**生成快速代码；
- 如何使用类型化（typed）内存视图高效地操作数组；
- 优化粒子模拟器；
- 有关在 Jupyter notebook 中使用 Cython 的提示；
- Cython 的剖析工具。

虽然懂点 C 语言会有所帮助，但本章只从 Python 优化的角度介绍 Cython，因此读者不需要具备任何 C 语言知识。

## 4.1 编译 Cython 扩展

Cython 语法被设计成 Python 语法的超集。在不做任何修改的情况下，Cython 就能够编译大部分 Python 模块（例外的情况不多）。Cython 源代码文件的扩展名为.pyx，可使用命令 `cython` 编译成 C 语言文件。

这里要介绍的第一个 Cython 脚本包含一个打印 `Hello, World!`的简单函数。

请新建一个名为 hello.pyx 的文件，并在其中输入如下代码。

```
def hello():
  print('Hello, World!')
```

下面的 `cython` 命令读取文件 hello.pyx，并生成文件 hello.c。

```
$ cython hello.pyx
```

为将 hello.c 编译成 Python 扩展模块，我们将使用编译器 GCC。我们需要添加一些 Python 专用的编译选项，这些选项因操作系统而异。必须指定包含头文件的目录，在下面的示例中，这个目录为/usr/include/python3.5/。

```
$ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -fno-strict-aliasing -lm -
I/usr/include/python3.5/ -o hello.so hello.c
```

要获悉 Python 的包含（include）目录，可使用 distutils 工具 sysconfig.get_python_inc。要执行这个工具，只需执行命令 python -c "from distutils import sysconfig; print(sysconfig.get_python_inc())"即可。

这将生成一个名为 hello.so 的文件，一个可直接在 Python 会话中导入的 C 语言扩展模块。

```
>>> import hello
>>> hello.hello()
Hello, World!
```



Cython 支持将 Python 2 和 Python 3 作为输入和输出语言，换而言之，要编译 Python 3 脚本文件 hello.pyx，可使用选项-3。

```
$ cython -3 hello.pyx
```

对于生成的 hello.c，无须做任何修改就可将其编译为 Python 2 或 Python 3 扩展模块，为此只需使用选项-I 指定相应的头文件即可，如下所示。

```
$ gcc -I/usr/include/python3.5 # ... other options
$ gcc -I/usr/include/python2.7 # ... other options
```

distutils 是标准的 Python 打包工具，使用它来编译 Cython 程序更简单。通过编写一个 setup.py 脚本，就可将.pyx 文件直接编译成扩展模块。例如，要编译前面的示例文件 hello.pyx，可编写一个最简单的 setup.py 脚本，它包含如下代码。

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
  name='Hello',
  ext_modules = cythonize('hello.pyx')
)
```

在上述代码中，开头两行导入函数 setup 和辅助函数 cythonize。调用函数 setup 时，传入了几个键–值对，它们指定了应用程序的名称以及需要创建的扩展。

辅助函数 cythonize 接受一个字符串或字符串列表，其中包含要编译的 Cython 模块。你也可以使用 glob 模式，如下面的代码所示。

```
cythonize(['hello.pyx', 'world.pyx', '*.pyx'])
```

要使用 distutils 编译前述扩展模块，可使用下面的代码执行脚本 setup.py。

```
$ python setup.py build_ext --inplace
```

选项 build_ext 让脚本 setup.py 构建 ext_modules 中指定的扩展模块,而选项--inplace 让这个脚本将输出文件 hello.so 放在源文件所在的目录（而不是构建目录）中。

你还可使用 pyximport 来自动编译 Cython 模块，为此只需在脚本开头调用 pyximport.install()即可（也可在解释器中执行这个命令）。这样做后，你就可直接导入.pyx 文件，而 pyximport 将透明地编译相应的 Cython 模块。

```
>>> import pyximport
>>> pyximport.install()
>>> import hello # 这将编译 hello.pyx
```

遗憾的是，并非在所有的配置下 pyximport 都管用（例如，同时涉及 C 和 Cython 文件时就不管用），但对测试简单脚本而言，这个工具很方便。

从 0.13 版起，IPython 就包含了 cythonmagic 扩展，让你能够交互地编写并测试一系列 Cython 语句。在 IPython shell 中，可使用 load_ext 来加载扩展：

```
%load_ext cythonmagic
```

加载这个扩展后，你就可使用**单元格魔法**命令%%cython 来编写多行的 Cython 代码片段。在下面的示例中，我们定义了函数 hello_snippet，它被编译并加入到 IPython 会话命名空间中。

```
%%cython
def hello_snippet():
    print("Hello, Cython!")

hello_snippet()
Hello, Cython!
```

## 4.2　添加静态类型

在 Python 中，在程序执行期间，变量可关联到不同类型的对象。这很好，因为它让这种语言灵活而动态，但也给解释器带来了很大的负担，因为解释器必须在运行阶段确定变量的类型及其包含的方法，这让很多优化都难以进行。Cython 扩展了 Python 语言，它支持显式的类型声明，因此能够通过编译生成高效的 C 语言扩展。

在 Cython 中，声明数据类型的主要方式是使用 cdef 语句。在多种情况下，都可使用关键字 cdef，如声明变量、函数和扩展类型（静态类）时。

### 4.2.1 变量

在 Cython 中，要声明变量的类型，可在变量名前加上关键字 `cdef` 和类型。例如，要将变量 `i` 声明为 16 位的整数，可像下面这样做。

```
cdef int i
```

在同一条 `cdef` 语句中，可声明多个变量，还可对变量进行初始化（这是可选的），如下面的代码所示。

```
cdef double a, b = 2.0, c = 3.0
```

对于类型化变量（typed variable），处理方式与常规变量不同。在 Python 中，变量通常被认为是指向内存中对象的**标签**，例如，可在程序的任何地方将值`'hello'`赋给变量 `a`。

```
a = 'hello'
```

这样，变量 `a` 将包含一个指向字符串`'hello'`的引用。在后续代码中，还可随便将一个其他的值（如整数 1）赋给这个变量。



```
a = 1
```

Python 将把整数 1 赋给变量 `a`，这不会有任何问题。

类型化变量的行为截然不同，它们通常被认为是**数据容器**：只能将适合的值存储到容器中，而是否适合取决于容器的数据类型。例如，如果我们将变量 `a` 声明为 `int` 类型，并试图将一个 `double` 值赋给它，Cython 将报错，如下面的代码所示。

```
%%cython
cdef int i
i = 3.0

# 对输出做了删节
...cf4b.pyx:2:4 Cannot assign type 'double' to 'int'
```

静态类型让编译器很容易执行有帮助的优化。例如，如果我们将一个循环索引声明为 `int` 类型，Cython 将使用纯粹的 C 代码重写循环，这样就不需要依赖于 Python 解释器。类型声明确保这个索引的类型始终为 `int`，在运行期间也不会改变，因此编译器可随便进行优化，而不会导致程序不再正确。

我们可使用一个小小的测试用例来评估这样速度将提高多少。在下面的示例中，我们实现了一个简单的循环，它将一个变量递增 100 次。使用 Cython 时，这个函数可这样编写：

```
%%cython
def example():
    cdef int i, j=0
    for i in range(100):
        j += 1
```

```
    return j

example()
# 结果：
# 100
```

我们可将其同一个类似的纯粹的 Python 循环的速度进行比较。

```
def example_python():
    j=0
    for i in range(100):
        j += 1
    return j

%timeit example()
10000000 loops, best of 3: 25 ns per loop
%timeit example_python()
100000 loops, best of 3: 2.74 us per loop
```

仅仅通过添加类型声明，速度就提高了令人惊讶的 100 倍！之所以会这样，是因为 Cython 循环首先被转换为纯粹的 C 语言代码，再被转换为高效的机器代码，而 Python 循环依赖于速度缓慢的解释器。

在 Cython 中，可将变量声明为任何标准的 C 语言类型，还可使用经典的 C 语言结构（如 `struct`、`enum` 和 `typedef`）来定义自定义类型。

一个有趣的例子是，如果我们将变量声明为 `object`，就可将任何类型的 Python 对象赋给它。

```
cdef object a_py
# 'hello'和 1 都是 Python 对象
a_py = 'hello'
a_py = 1
```

请注意，将变量的类型声明为 `object` 对性能提升没有任何好处，因为访问和操作对象时，也将要求解释器确定变量的类型及其包含的属性和方法。

在有些情况下，有些数据类型（如 `float` 和 `int`）是兼容的，因此可在它们之间相互转换。在 Cython 中，要进行类型转换（**强制转换**），可在尖括号内指定目标类型，如下面的代码所示。

```
cdef int a = 0
cdef double b
b = <double> a
```

### 4.2.2　函数

要给 Python 函数的参数添加类型信息，可在参数名前面指定类型。这样定义的函数的行为与常规 Python 函数相同，但将对其参数执行类型检查。我们可编写一个名为 `max_python` 的函数，它返回两个整数中较大的那个。

```
def max_python(int a, int b):
    return a if a > b else b
```

对于这样定义的函数，将执行类型检查，并将其参数视为类型化变量，就像是在 cdef 语句中定义的一样。然而，这样的函数依然是 Python 函数，多次调用时依然需要切换到解释器。要让 Cython 能够优化函数调用，必须使用 cdef 语句声明函数的返回类型。

```
cdef int max_cython(int a, int b):
    return a if a > b else b
```

这样声明的函数将被转换为原生 C 语言函数，其开销比 Python 函数低得多。一个重大缺陷是，这样的函数不能在 Python 中使用，而只能在 Cython 中使用。同时，它们的作用域为当前 Cython 文件——除非在一个定义文件中暴露它们（参见后面的 4.3 节）。

所幸 Cython 允许你定义可在 Python 中调用且可转换为高性能 C 语言函数的函数。如果你使用 cpdef 语句定义一个函数，Cython 将生成这个函数的两个版本：可供解释器使用的 Python 版本；可在 Cython 中使用的快速的 C 语言函数。cpdef 语句的语法与 cdef 语句相同，如下所示。

4

```
cpdef int max_hybrid(int a, int b):
    return a if a > b else b
```

在有些情况下，即便是 C 语言函数，调用开销从性能上说也是个问题，函数在关键循环中被调用很多次时尤其如此。在函数体很短时，最好在函数定义前加上关键字 inline，这样函数调用将被替换为函数体本身。前述返回最大值的函数就非常适合声明为**内联**的。

```
cdef inline int max_inline(int a, int b):
    return a if a > b else b
```

### 4.2.3 类

要声明扩展类型，可使用 cdef class 语句，并在类体中声明属性。例如，我们可创建一个名为 Point 的扩展类型，它存储两个类型为 double 的坐标$(x, y)$，如下面的代码所示。

```
cdef class Point
    cdef double x
    cdef double y
def __init__(self, double x, double y):
    self.x = x
    self.y = y
```

在类方法中访问声明的属性时，Cython 将绕过开销很大的属性查找，直接访问底层 C 语言结构体中的指定字段。有鉴于此，访问类型化类的属性的速度极快。

要在代码中使用 cdef class，需要显式地声明要在编译期间使用的变量的类型。在任何可使用标准类型（如 double、float 和 int）的地方，都可使用扩展类型（如 Point）。例如，如果你要编写一个 Cython 函数（在下面的示例中，这个函数名为 norm），它计算一个点到原点

的距离，就必须将输入变量的类型声明为 `Point`，如下面的代码所示。

```
cdef double norm(Point p):
    return (p.x**2 + p.y**2)**0.5
```

与类型化函数一样，类型化类也有一些限制。如果你在 Python 中试图访问扩展类型的属性，将引发 `AttributeError` 异常，如下所示。

```
>>> a = Point(0.0, 0.0)
>>> a.x
AttributeError: 'Point' object has no attribute 'x'
```

要在 Python 代码中访问属性，必须在属性声明中使用限定符 `public`（可读写）或 `readonly`，如下面的代码所示。

```
cdef class Point:
    cdef public double x
```

另外，要声明方法，可使用 `cpdef` 语句，就像声明常规函数一样。

对于扩展类型，不能在运行阶段给它添加额外的属性。要这样做，一种解决方案是从类型化类派生出一个 Python 类，并使用纯粹的 Python 来扩展其属性和方法。

## 4.3 共享声明

编写 Cython 模块时，你可能想重新组织最常用的函数和类声明，将它们放在一个独立的文件中，以便在不同的模块中重用。在 Cython 中，可将这些声明放在**定义文件**中，并使用 `cimport` 语句来访问它们。

假设有一个模块，其中包含函数 `max` 和 `min`，而我们想在多个 Cython 程序中重用它们。如果在一个.pyx 文件中编写一系列函数，这些声明将只能在该文件中使用。

> 定义文件还被用来建立 Cython 到外部 C 语言代码的接口，其中的理念是将类型和函数原型复制（更准确地说是转移）到定义文件中，并将实现保留在将单独编译和链接的外部 C 语言代码中。

为共享函数 `max` 和 `min`，需要编写一个扩展名为.pxd 的定义文件。这种文件只包含要与其他模块共享的类型和函数原型，即相当于是一个**公有**接口。我们可在一个名为 mathlib.pxd 的文件中声明函数 `max` 和 `min` 的原型，如下所示。

```
cdef int max(int a, int b)
cdef int min(int a, int b)
```

如你所见，我们只编写了函数名和参数，而没有实现函数体。

函数实现将放在实现文件中，该实现文件的文件名与定义文件相同，但扩展名为.pyx。换句话说，这个实现文件名为 mathlib.pyx。

```
cdef int max(int a, int b):
  return a if a > b else b

cdef int min(int a, int b):
  return a if a < b else b
```

现在可以在另一个 Cython 模块中导入模块 `mathlib` 了。

为测试这个新建的 Cython 模块，我们将创建一个名为 distance.pyx 的文件，其中包含一个名为 `chebyshev` 的函数。这个函数计算两个点之间的切比雪夫距离，如下面的代码所示。两组坐标`(x1, y1)`和`(x2, y2)`之间的切比雪夫距离指的是对应坐标的最大差值。

```
max(abs(x1 - x2), abs(y1 - y2))
```

我们将使用 `cimport` 导入 mathlib.pxd 中声明的函数 `max`，以便使用它来实现函数 `chebyshev`，如下面的代码所示。

```
from mathlib cimport max

def chebyshev(int x1, int y1, int x2, int y2):
    return max(abs(x1 - x2), abs(y1 - y2))
```

其中的 `cimport` 语句将读取 mathlib.pxd，而生成文件 distance.c 时，将用到函数 `max` 的定义。

## 4.4 使用数组

高性能数值计算常常要用到数组。Cython 提供了一种与数组交互的简单方式：直接使用低级 C 语言数组或更通用的**类型化内存视图**。

### 4.4.1 C 语言数组和指针

C 语言数组是一系列类型相同的元素，这些元素在内存中存储在一起。深入其中的细节前，弄明白（或复习一下）C 语言是如何管理内存的将大有裨益。

在 C 语言中，变量犹如容器。当你创建变量时，将在内存中预留空间，用于存储变量的值。例如，如果你创建一个用于存储 64 位浮点数（即类型为 `double`）的变量，程序将分配 64 位（即 16 字节）内存。这部分内存可通过指向它的地址来访问。

要获取变量的地址，可使用**地址运算符**（符号`&`）。要打印变量的地址，可使用 Cython 模块 `libc.stdio` 中的函数 `printf`，如下所示。

```
%%cython
cdef double a
from libc.stdio cimport printf
printf("%p", &a)
# 输出：
# 0x7fc8bb611210
```

内存地址可存储在被称为**指针**的特殊变量中，而要声明指针，可在变量名前面加上前缀*，如下所示。

```
from libc.stdio cimport printf
cdef double a
cdef double *a_pointer
a_pointer = &a # a_pointer 和&a 的类型相同
```

要获取指针指向的地址中存储的值，可使用**解除引用运算符**（符号*）。请注意，在这种情况下，*的含义与变量声明中的*不同。

```
cdef double a
cdef double *a_pointer
a_pointer = &a

a = 3.0
print(*a_pointer) # 打印 3.0
```

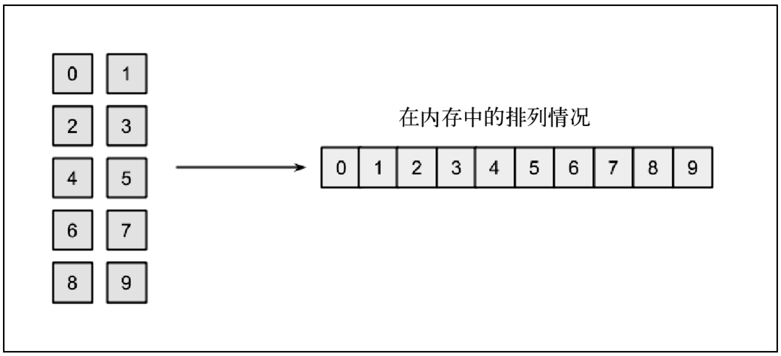
当你声明 C 语言数组时，程序将分配足够的空间，以便存储指定数量的元素。例如，当你声明一个包含 10 个元素的 double 数组（每个元素需要 16 字节）时，程序将在内存中预留 160（16 × 10）字节的连续空间。在 Cython 中，要声明这样的数组，可使用下面的语法。

```
cdef double arr[10]
```

你还可声明多维数组，如 5 行 2 列的数组。为此可使用如下语法：

```
cdef double arr[5][2]
```

这将分配一块连续的内存，一行接一行。这种顺序被称为**行主序**（row-major），如下图所示。数组也可能是**列主序**（column-major）的，在编程语言 FORTRAN 中就是这样的。



> TIP 数组元素的排列顺序有重要影响。在最后一维上迭代 C 语言数组时，访问的是连续的内存块（在前面的示例中，将依次访问 0、1、2、3 等），而在第一维上迭代时，每次都将跳过一些位置（在前面的示例中，将依次访问 0、2、4、6、8、1 等）。在任何情况下，都应尽可能依次访问内存，因为这样可优化缓存和内存使用情况。

要获取或修改数组元素，可使用标准索引；C 语言数组不支持花式索引和切片。

```
arr[0] = 1.0
```

C 语言数组的有些行为与指针相同。实际上，变量 arr 指向的是相应数组的第一个元素所在的内存单元。为验证数组的第一个元素的地址与变量 arr 包含的地址相同，可使用地址运算符，如下所示。

```
%%cython
from libc.stdio cimport printf
cdef double arr[10]
printf("%pn", arr)
printf("%pn", &arr[0])

# 输出：
# 0x7ff6de204220
# 0x7ff6de204220
```



需要建立到既有 C 语言库的接口或需要细致地控制内存时，应使用 C 语言数组和指针，而且它们的性能都非常高。但这样细致的控制也容易出错，因为它无法防止你访问错误的内存单元。对于更常见的用例，为提高安全性，可使用 NumPy 数组或类型化内存视图。

### 4.4.2 NumPy 数组

在 Cython 中，可将 NumPy 数组作为常规 Python 对象使用，以利用它们经过优化的广播操作。然而，Cython 提供了一个名为 numpy 的模块，这个模块提供了更强的直接迭代支持。

当你以常规方式访问 NumPy 数组的元素时，在解释器层面将执行其他一些操作，这将带来很大的开销。Cython 可避开这些操作和检查，直接操作 NumPy 数组使用的内存区域，从而极大地改善性能。

要声明 NumPy 数组，可使用数据类型 ndarray，而要在代码中使用这种数据类型，必须先使用 cimport 导入 Cython 模块 numpy（它不同于 Python 模块 numpy）。我们将把这个模块绑定到变量 c_np，以便将其与 Python 模块 numpy 区分开来。

```
cimport numpy as c_np
import numpy as np
```

现在可以声明 NumPy 数组了。方法是在方括号内指定类型和维数，这被称为**缓冲区语法**

（buffer syntax）。要声明一个二维的 double 数组，可使用如下代码：

```
cdef c_np.ndarray[double, ndim=2] arr
```

访问这个数组时，将直接操作底层的内存区域，从而极大地提升速度。

在下面的示例中，我们将演示如何使用类型化 numpy 数组，并将其与常规 Python 版本进行比较。

我们首先编写函数 numpy_bench_py，它将 py_arr 的每个元素都加 1。我们将索引 i 的类型声明为 int，以消除 for 循环的开销。

```
%%cython
import numpy as np
def numpy_bench_py():
    py_arr = np.random.rand(1000)
    cdef int i
    for i in range(1000):
        py_arr[i] += 1
```

接下来，我们使用类型 ndarray 编写同样的函数。请注意，使用 c_np.ndarray 声明变量 c_arr 后，就可将一个使用 Python 模块 numpy 创建的数组赋给它。

```
%%cython
import numpy as np
cimport numpy as c_np

def numpy_bench_c():
    cdef c_np.ndarray[double, ndim=1] c_arr
    c_arr = np.random.rand(1000)
    cdef int i

    for i in range(1000):
        c_arr[i] += 1
```

现在可以使用 timeit 测量这两个函数的执行时间了。从测量结果可知，类型化版本的速度快了 50 倍。

```
%timeit numpy_bench_c()
100000 loops, best of 3: 11.5 us per loop
%timeit numpy_bench_py()
1000 loops, best of 3: 603 us per loop
```

### 4.4.3　类型化内存视图

C 数组和 NumPy 数组与内置对象 bytes、bytearray 和 array.array 很像，因为它们都在连续的内存区域（也叫**内存缓冲区**）上操作。Cython 提供了一个通用接口——**类型化内存视图**，该接口统一并简化了对所有这些数据类型的访问。

**内存视图**是一个对象，维护着一个指向特定内存区域的引用。该内存区域实际上并不归内存视图所有，但内存视图能够读取和修改其内容；换而言之，内存视图是一个有关底层数据的**视图**。要定义内存视图，可使用一种特殊语法。例如，要定义一个 int 内存视图和一个二维的 double 内存视图，可像下面这样做。

```
cdef int[:] a
cdef double[:, :] b
```

这种语法也可用于声明任何类型的变量和类属性，还可用于函数定义中。任何暴露了缓冲区接口的对象（如 NumPy 数组、bytes 和 array.array）都将自动绑定到内存视图。例如，可使用简单的变量赋值将一个 NumPy 数组绑定到内存视图。

```
import numpy as np

cdef int[:] arr
arr_np = np.zeros(10, dtype='int32')
arr = arr_np # 将数组绑定到内存视图
```



必须指出的是，内存视图并不**拥有**与之绑定的数据，而只是提供了一种**访问**和**修改**它们的途径。在这个示例中，数据归 NumPy 数组所有。从下面的示例可知，通过内存视图修改数据时，操作的是底层的内存区域，因此这种修改将在原始 NumPy 数组中反映出来（反之亦然）。

```
arr[2] = 1 # 修改内存视图
print(arr_np)
# [0 0 1 0 0 0 0 0 0 0]
```

从某种意义上说，内存视图的效果类似于对 NumPy 数组执行切片操作。第 3 章介绍过，对 NumPy 数组执行切片操作时，不会复制数据，而是返回一个指向相应内存区域的视图，而对该视图所做的修改将在原始数组中反映出来。

对于内存视图，也可使用标准的 NumPy 语法来执行切片操作。

```
cdef int[:, :, :] a
arr[0, :, :] # 一个二维的内存视图
arr[0, 0, :] # 一个一维的内存视图
arr[0, 0, 0] # 一个 int 值
```

要在内存视图之间复制数据，可使用类似于切片赋值的语法，如下面的代码所示。

```
import numpy as np

cdef double[:, :] b
cdef double[:] r
b = np.random.rand(10, 3)
r = np.zeros(3, dtype='float64')

b[0, :] = r # 将 r 的值复制到 b 的第 1 行中
```

在下一节中，我们将在粒子模拟器中使用类型化内存视图来声明数组的类型。

## 4.5　使用 Cython 编写粒子模拟器

对 Cython 的工作原理有大致了解后，便可重写方法 ParticleSimulator.evolve 了。多亏了 Cython，我们能够将这些循环转换为 C 语言的，从而消除 Python 解释器带来的开销。

在第 3 章，我们使用 NumPy 编写了方法 evolve，其效率相当高。我们将这个旧版本重命名为 evolve_numpy，以便与新版本区分开来。

```
def evolve_numpy(self, dt):
    timestep = 0.00001
    nsteps = int(dt/timestep)

    r_i = np.array([[p.x, p.y] for p in self.particles])
    ang_speed_i = np.array([p.ang_speed for p in self.particles])
    v_i = np.empty_like(r_i)

    for i in range(nsteps):
        norm_i = np.sqrt((r_i ** 2).sum(axis=1))

        v_i = r_i[:, [1, 0]]
        v_i[:, 0] *= -1
        v_i /= norm_i[:, np.newaxis]

        d_i = timestep * ang_speed_i[:, np.newaxis] * v_i

        r_i += d_i

    for i, p in enumerate(self.particles):
        p.x, p.y = r_i[i]
```

我们要将这些代码转换为 Cython 的。我们将消除 NumPy 数组广播，将这个方法转换为一个基于索引的算法，以利用快速的索引操作。Cython 生成的是高效的 C 语言代码，因此我们想使用多少循环就可使用多少，而不会对性能有任何影响。

作为一种设计选择，我们在一个函数中重写循环，并将这个函数放在一个名为 cevolve.pyx 的 Cython 模块中。这个模块只包含一个 Python 函数——c_evolve，而这个函数将粒子位置、角速度、步长和步数作为参数。

一开始，我们不添加类型信息，而只将这个函数隔离，并确保编译其所在的模块时不会出错。

```
# 文件：simul.py
def evolve_cython(self, dt):
    timestep = 0.00001
    nsteps = int(dt/timestep)

    r_i = np.array([[p.x, p.y] for p in self.particles])
    ang_speed_i = np.array([p.ang_speed for p in self.particles])

    c_evolve(r_i, ang_speed_i, timestep, nsteps)
```

```
        for i, p in enumerate(self.particles):
            p.x, p.y = r_i[i]

# 文件：cevolve.pyx
import numpy as np

def c_evolve(r_i, ang_speed_i, timestep, nsteps):
    v_i = np.empty_like(r_i)

    for i in range(nsteps):
        norm_i = np.sqrt((r_i ** 2).sum(axis=1))

        v_i = r_i[:, [1, 0]]
        v_i[:, 0] *= -1
        v_i /= norm_i[:, np.newaxis]

        d_i = timestep * ang_speed_i[:, np.newaxis] * v_i

        r_i += d_i
```



请注意，函数 c_evolve 无须返回值，因为它就地修改了数组 r_i 中的值。为对 NumPy 版本和未指定类型信息的 Cython 版本进行基准测试，可稍微修改一下函数 benchmark，如下所示。

```
def benchmark(npart=100, method='python'):
    particles = [Particle(uniform(-1.0, 1.0),
                          uniform(-1.0, 1.0),
                          uniform(-1.0, 1.0))
                          for i in range(npart)]
    simulator = ParticleSimulator(particles)
    if method=='python':
        simulator.evolve_python(0.1)
    elif method == 'cython':
        simulator.evolve_cython(0.1)
    elif method == 'numpy':
        simulator.evolve_numpy(0.1)
```

现在可以在 IPython shell 中测量不同版本的执行时间了。

```
%timeit benchmark(100, 'cython')
1 loops, best of 3: 401 ms per loop
%timeit benchmark(100, 'numpy')
1 loops, best of 3: 413 ms per loop
```

这两个版本的速度相同，这表明相比于纯粹的 Python 代码，编译没有指定静态类型信息的 Cython 模块没有任何优势可言。接下来，对于所有重要的变量，我们都声明其类型，让 Cython 能够进行优化。

首先，我们给函数参数声明类型，看看性能有何变化。对于数组参数，我们将其类型声明为包含 double 值的内存视图。需要指出的是，如果我们在调用这个函数时传入 int 或 float32 数组，将不会自动进行类型转换，因此将出错。

```
def c_evolve(double[:, :] r_i,
             double[:] ang_speed_i,
             double timestep,
             int nsteps):
```

现在可以重写分多步处理粒子的循环了。在这个循环中，我们可将迭代索引 `i` 和 `j` 以及表示粒子数量的 `nparticles` 都声明为 `int` 类型。

```
cdef int i, j
cdef int nparticles = r_i.shape[0]
```

这里使用的算法与纯粹的 Python 版很像：分多步迭代粒子，并计算每个粒子坐标的速度和位移向量，如下面的代码所示。

```
for i in range(nsteps):
    for j in range(nparticles):
        x = r_i[j, 0]
        y = r_i[j, 1]
        ang_speed = ang_speed_i[j]

        norm = sqrt(x ** 2 + y ** 2)

        vx = (-y)/norm
        vy = x/norm

        dx = timestep * ang_speed * vx
        dy = timestep * ang_speed * vy

        r_i[j, 0] += dx
        r_i[j, 1] += dy
```

在上述代码中，我们添加了变量 `x`、`y`、`ang_speed`、`norm`、`vx`、`vy`、`dx` 和 `dy`。为避免使用 Python 解释器带来的开销，我们必须在函数开头声明这些变量的类型，如下所示。

```
cdef double norm, x, y, vx, vy, dx, dy, ang_speed
```

我们还使用了一个名为 `sqrt` 的函数来计算 norm。如果使用模块 `math` 或 `numpy` 中的 `sqrt`，这个重要的循环将包含一个速度很慢的 Python 函数，进而影响代码的性能。在标准 C 语言库中，有一个速度很快的 `sqrt` 函数，它被封装在 Cython 模块 `libc.math` 中。

```
from libc.math cimport sqrt
```

现在可再次运行基准测试程序，看看性能改善情况了，如下所示。

```
In [4]: %timeit benchmark(100, 'cython')
100 loops, best of 3: 13.4 ms per loop
In [5]: %timeit benchmark(100, 'numpy')
1 loops, best of 3: 429 ms per loop
```

在粒子数量很少的情况下，速度得到了极大的提升，达到了以前版本的 40 倍。然而，我们

还应测试粒子数量更多时的性能情况。

```
In [2]: %timeit benchmark(1000, 'cython')
10 loops, best of 3: 134 ms per loop
In [3]: %timeit benchmark(1000, 'numpy')
1 loops, best of 3: 877 ms per loop
```

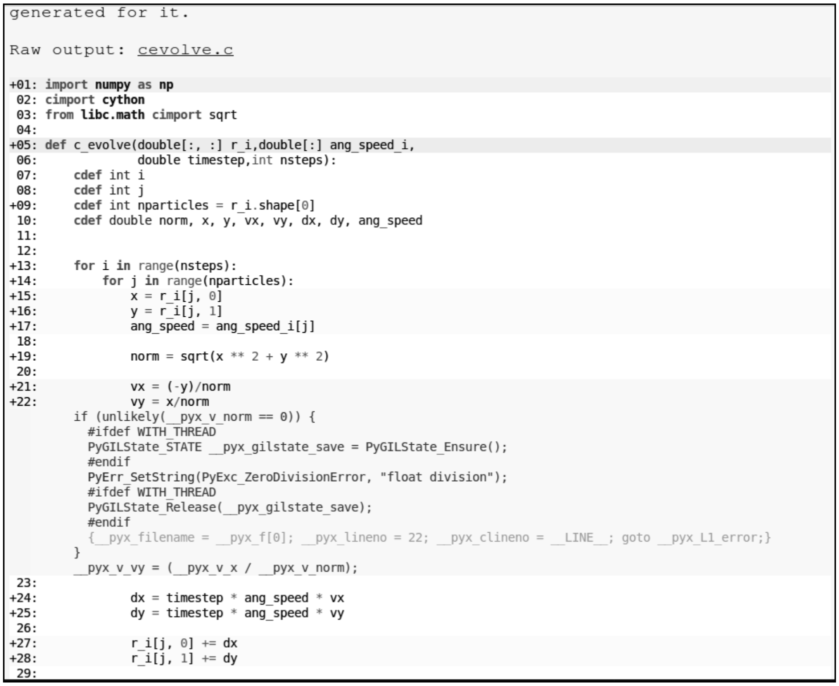
随着粒子数量的增加，两个版本的速度更为接近。将粒子数量增加到 1000 个后，性能提升降低到了 6 倍。这可能是因为随着粒子数量的增加，Python `for` 循环的开销相比于其他操作的开销越来越小。

## 4.6 剖析 Cython 代码

Cython 提供了一种名为**注释视图**（annotated view）的功能，让我们能够获悉哪些代码行是在 Python 解释器中执行的，以及哪些代码行存在很大的优化空间。要启用这种功能，可在编译 Cython 文件时指定选项`-a`，这样 Cython 将生成一个 HTML 文件，其中包含 Cython 代码以及一些很有用的注释信息。选项`-a` 的用法如下：

```
$ cython -a cevolve.pyx
$ firefox cevolve.html
```

生成的 HTML 文件如下面的屏幕截图所示，它逐行地显示了 Cython 文件的内容。



```
generated for it.

Raw output: cevolve.c

+01: import numpy as np
 02: cimport cython
 03: from libc.math cimport sqrt
 04:
+05: def c_evolve(double[:, :] r_i,double[:] ang_speed_i,
 06:                  double timestep,int nsteps):
 07:     cdef int i
 08:     cdef int j
+09:     cdef int nparticles = r_i.shape[0]
 10:     cdef double norm, x, y, vx, vy, dx, dy, ang_speed
 11:
 12:
+13:     for i in range(nsteps):
+14:         for j in range(nparticles):
+15:             x = r_i[j, 0]
+16:             y = r_i[j, 1]
+17:             ang_speed = ang_speed_i[j]
 18:
+19:             norm = sqrt(x ** 2 + y ** 2)
 20:
+21:             vx = (-y)/norm
+22:             vy = x/norm
        if (unlikely(__pyx_v_norm == 0)) {
          #ifdef WITH_THREAD
          PyGILState_STATE __pyx_gilstate_save = PyGILState_Ensure();
          #endif
          PyErr_SetString(PyExc_ZeroDivisionError, "float division");
          #ifdef WITH_THREAD
          PyGILState_Release(__pyx_gilstate_save);
          #endif
          {__pyx_filename = __pyx_f[0]; __pyx_lineno = 22; __pyx_clineno = __LINE__; goto __pyx_L1_error;}
        }
        __pyx_v_vy = (__pyx_v_x / __pyx_v_norm);
 23:
+24:             dx = timestep * ang_speed * vx
+25:             dy = timestep * ang_speed * vy
 26:
+27:             r_i[j, 0] += dx
+28:             r_i[j, 1] += dy
 29:
```

每行源代码都可能带有深度不同的黄色背景。背景色越深，表明代码与解释器调用的相关程度越高；而背景色为白色的代码将被转换为常规 C 语言代码。由于解释器调用会极大地降低执行速度，因此我们的目标是让函数体内代码的背景色尽可能浅。对于任何代码行，都可通过单击它来查看 Cython 编译器生成的代码。例如，代码行 `v_y = x/norm` 核实 norm 不为 0，如果这个条件不满足，将引发 `ZeroDivisionError` 异常。对于代码行 `x = r_i[j, 0]`，Cython 将检查这些索引是否在数组的范围内。你可能注意到了，最后一行的背景色很深，但通过查看代码可知，这实际上是个误会：这行代码对应的是与函数末尾相关的模板代码。

Cython 可禁用检查（如检查除数是否为零），从而删除这些与解释器相关的调用。这通常是通过编译器指令实现的。添加编译器指令的方式有多种：

- 使用装饰器或上下文管理器；
- 在文件开头使用注释；
- 使用 Cython 命令行选项。

完整的 Cython 编译器指令清单，请参阅官方文档，其网址为 http://docs.cython.org/src/reference/compilation.html#compiler-directives。

例如，要禁用数组边界检查，只需像下面这样使用 `cython.boundscheck` 对函数进行装饰即可。

```
cimport cython

@cython.boundscheck(False)
def myfunction():
    # 函数的代码
```

也可像下面这样使用 `cython.boundscheck` 将代码块封装在上下文管理器中：

```
with cython.boundscheck(False):
    # 代码块
```

要在整个模块中禁用边界检查，可在文件开头添加如下代码行：

```
# cython: boundscheck=False
```

要使用命令行选项修改编译器指令，可像下面这样使用选项`-X`：

```
$ cython -X boundscheck=True
```

要在函数 `c_evolve` 中禁用额外的检查，可禁用编译器指令 `boundscheck` 并启用编译器指令 `cdivision`（这样将不检查 `ZeroDivisionError`），如下面的代码所示。

```
cimport cython

@cython.boundscheck(False)
```

```
@cython.cdivision(True)
def c_evolve(double[:, :] r_i,
             double[:] ang_speed_i,
             double timestep,
             int nsteps):
```

这样做后，如果再次查看注释视图，将发现整个循环体的背景都是白色了——解释器已不再涉足内部循环。要重新编译这些代码，只需再次执行命令 `python setup.py build_ext --inplace` 即可。但再次运行基准测试程序后，我们发现性能并没有得到改善，这表明这些检查并非瓶颈的一部分。

```
In [3]: %timeit benchmark(100, 'cython')
100 loops, best of 3: 13.4 ms per loop
```

另一种剖析 Cython 代码的方式是使用模块 `cProfile`。例如，我们可以编写一个简单的函数，它计算两个坐标数组之间的切比雪夫距离。为此，请创建一个名为 cheb.py 的文件，如下所示。



```
import numpy as np
from distance import chebyshev

def benchmark():
    a = np.random.rand(100, 2)
    b = np.random.rand(100, 2)
    for x1, y1 in a:
        for x2, y2 in b:
            chebyshev(x1, x2, y1, y2)
```

如果对这个脚本进行剖析，将得不到有关前面使用 Cython 实现的函数的统计信息。要收集有关函数 `max` 和 `min` 的剖析信息，需要在文件 mathlib.pyx 开头添加选项 `profile=True`，如下面的代码所示。

```
# cython: profile=True

cdef int max(int a, int b):
    # 其他代码
```

现在可以在 IPython 中使用`%prun` 来剖析这个脚本了，如下所示。

```
    import cheb
    %prun cheb.benchmark()

# 输出:
2000005 function calls in 2.066 seconds

  Ordered by: internal time
  ncalls tottime percall cumtime percall filename:lineno(function)
       1   1.664   1.664   2.066   2.066 cheb.py:4(benchmark)
 1000000   0.351   0.000   0.401   0.000 {distance.chebyshev}
 1000000   0.050   0.000   0.050   0.000 mathlib.pyx:2(max)
```

```
        2    0.000    0.000    0.000    0.000 {method 'rand' of
'mtrand.RandomState' objects}
        1    0.000    0.000    2.066    2.066 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
```

上述输出包含有关函数 `max` 的信息，这些信息表明这个函数并非瓶颈。大部分时间都花在函数 `benchmark` 上，这意味着瓶颈很可能是纯粹的 Python `for` 循环。就这个示例而言，最佳策略是使用 NumPy 重写这个循环，或者将代码移植到 Cython。

## 4.7　在 Jupyter 中使用 Cython

要优化 Cython 代码，必须反复尝试。所幸通过 Jupyter notebook 可方便地访问 Cython 工具，这提供了更便利、更和谐的优化体验。

要启动 notebook 会话，可在命令行中执行命令 `jupyter notebook`；要加载 Cython 魔法命令，可在单元格中输入`%load_ext cython`。

前面说过，要在当前会话中编译并加载 Cython 代码，可使用魔法命令`%%cython`。例如，要在单元格中复制 cheb.py 的内容，可像下面这样做。

```
%%cython
import numpy as np

cdef int max(int a, int b):
    return a if a > b else b

cdef int chebyshev(int x1, int y1, int x2, int y2):
    return max(abs(x1 - x2), abs(y1 - y2))

def c_benchmark():
    a = np.random.rand(1000, 2)
    b = np.random.rand(1000, 2)

    for x1, y1 in a:
        for x2, y2 in b:
            chebyshev(x1, x2, y1, y2)
```

魔法命令`%%cython` 提供了很有用的选项`-a`,让你能够在 notebook 中直接编译代码并生成其注释视图（就像命令行选项`-a` 一样），如下面的屏幕截图所示。

```
In [15]: %%cython -a
         import numpy as np

         cdef int max(int a, int b):
             return a if a > b else b

         cdef int chebyshev(int x1, int y1, int x2, int y2):
             return max(abs(x1 - x2), abs(y1 - y2))

         def c_benchmark():
             a = np.random.rand(1000, 2)
             b = np.random.rand(1000, 2)

             for x1, y1 in a:
                 for x2, y2 in b:
                     chebyshev(x1, x2, y1, y2)

Out[15]:
Generated by Cython 0.25.2

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.
 01: # cython: profile=True
+02: import numpy as np
 03:
+04: cdef int max(int a, int b):
+05:     return a if a > b else b
 06:
+07: cdef int chebyshev(int x1, int y1, int x2, int y2):
+08:     return max(abs(x1 - x2), abs(y1 - y2))
 09:
+10: def c_benchmark():
+11:     a = np.random.rand(1000, 2)
+12:     b = np.random.rand(1000, 2)
 13:
+14:     for x1, y1 in a:
+15:         for x2, y2 in b:
+16:             chebyshev(x1, x2, y1, y2)
```

这让你能够快速测试代码的不同版本以及使用 Jupyter 中的其他集成工具。例如，要在当前会话中测量代码的执行时间和剖析代码，可分别使用工具`%timeit`和`%prun`（条件是在单元格中启用了编译器指令`profile`）。下面的屏幕截图演示了如何使用魔法命令`%prun`来查看剖析结果。

```
In [22]: %prun c_benchmark()

         2000005 function calls in 1.370 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    1.127    1.127    1.370    1.370 _cython_magic_c7d6eab16ab5658137c9af8534d5cafb.pyx:10(c_benchma
rk)
  1000000    0.191    0.000    0.243    0.000 _cython_magic_c7d6eab16ab5658137c9af8534d5cafb.pyx:7(chebyshev)
  1000000    0.052    0.000    0.052    0.000 _cython_magic_c7d6eab16ab5658137c9af8534d5cafb.pyx:4(max)
        1    0.000    0.000    1.370    1.370 <string>:1(<module>)
        1    0.000    0.000    1.370    1.370 {built-in method builtins.exec}
        1    0.000    0.000    1.370    1.370 {_cython_magic_c7d6eab16ab5658137c9af8534d5cafb.c_benchmark}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

你还可在 notebook 中直接使用第 1 章讨论的工具`line_profiler`。要支持行注释（line annotations），必须做如下工作：

- 启用编译指令`linetrace`和`binding`（将它们都设置为`True`）；
- 在编译阶段启用标志`CYTHON_TRACE`（将其设置为 1）。

要完成这些工作很容易，只需给魔法命令`%%cython`添加相应的参数，并在代码中设置相应的编译指令，如下所示。

```
%%cython -a -f -c=-DCYTHON_TRACE=1
# cython: linetrace=True
# cython: binding=True

import numpy as np

cdef int max(int a, int b):
    return a if a > b else b

def chebyshev(int x1, int y1, int x2, int y2):
    return max(abs(x1 - x2), abs(y1 - y2))

def c_benchmark():
    a = np.random.rand(1000, 2)
    b = np.random.rand(1000, 2)

    for x1, y1 in a:
        for x2, y2 in b:
            chebyshev(x1, x2, y1, y2)
```

准备工作完成后，就可使用魔法命令`%lprun`对代码进行剖析了。

```
%lprun -f c_benchmark c_benchmark()
# 输出:
Timer unit: 1e-06 s

Total time: 2.322 s
File:
/home/gabriele/.cache/ipython/cython/_cython_magic_18ad8204e9d29650f3b09feb
48ab0f44.pyx
Function: c_benchmark at line 11

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    11                                           def c_benchmark():
    12         1          226    226.0      0.0      a = np.random.rand...
    13         1           67     67.0      0.0      b = np.random.rand...
    14
    15      1001         1715      1.7      0.1      for x1, y1 in a:
    16   1001000      1299792      1.3     56.0          for x2, y2 in b:
    17   1000000      1020203      1.0     43.9              chebyshev...
```

如你所见，第 16 行花费了大量时间，这是一个纯粹的 Python 循环，很有必要进一步优化。

Jupyter 提供的工具让你能够快速完成编辑–编译–测试循环，从而快速创建原型并节省测试不同解决方案所需的时间。

## 4.8　小结

Cython 兼具 Python 的便利性和 C 语言的速度。相比于 C 绑定（binding），Cython 程序维护和调试起来要容易得多,这要归功于 Cython 与 Python 的紧密集成和兼容性以及一些卓越的工具。

本章介绍了 Cython 语言的基础知识，以及如何通过给变量和函数参数添加静态类型来提高程序的速度。你还学习了如何使用 C 语言数组、NumPy 数组和内存视图。

我们对粒子模拟器进行了优化：通过重写其中重要的函数 `evolve`，极大地提高了速度。最后学习了如何使用注释视图来找出原本难以找出的与解释器相关的调用，以及如何在 Cython 中启用对 `cProfile` 的支持。另外，还学习了如何使用 Jupyter 集成的工具来剖析和分析 Cython 代码。

下一章将探索其他一些工具，它们可动态地生成速度更快的机器码，而不要求你先将代码编译成 C 语言代码。

第 5 章

# 探索编译器

Python 是一款使用广泛而且成熟的语言，因此人们有很大的动力去改进它的性能，办法是直接将函数和方法编译成机器码，而不是在解释器中执行指令。第 4 章介绍了一个这样的例子，它通过声明类型、编译成高效的 C 代码并避免解释器调用来改善 Python 代码。

本章将探索两个项目——Numba 和 PyPy，它们以与 Cython 稍有不同的方式进行编译。Numba 是一个库，设计用于动态地编译小型函数。它不是将 Python 代码转换为 C 代码，而是对 Python 函数进行分析并将其直接编译成机器码。PyPy 是一款解释器，它在运行阶段对代码进行分析，并自动对速度缓慢的循环进行优化。

这些工具都被称为**即时**（just-in-time，JIT）**编译器**，因为编译是在运行阶段而不是运行代码前进行的［在运行代码前进行编译的编译器称为预先（ahead-of-time，AOT）编译器］。

本章介绍如下主题：

- Numba 基础；
- 使用原生模式编译实现快速函数；
- 理解并实现通用函数；
- JIT 类；
- 安装 PyPy；
- 使用 PyPy 运行粒子模拟器；
- 其他有趣的编译器。

## 5.1 Numba

Numba 是 2012 年面世的，出自 NumPy 最初的开发者 Travis Oliphant 之手。这是一个库，它在运行阶段使用**低级虚拟机**（low-level virtual machine，LLVM）工具链对 Python 函数进行编译。

LLVM 是一组设计用于编写编译器的工具，它并非针对特定语言的，因此被用来为众多的语言编写编译器（一个著名的例子是 clang 编译器）。LLVM 的一个核心方面是中间表示（intermediate

representation，即 LLVM IR），这是一种独立于平台的低级语言（类似于汇编语言），可通过对其进行编译来生成在特定平台上运行的机器码。

Numba 检查 Python 函数，并使用 LLVM 将其编译为 IR。正如你在前一章看到的，通过给变量和函数参数声明类型，可提升速度。Numba 实现了巧妙的类型猜测算法（这被称为类型推断），并通过编译包含类型信息的函数版本来提高执行速度。

请注意，开发 Numba 旨在改善执行数值计算的代码的性能，因此其重点是优化大量使用 NumPy 数组的应用程序。

Numba 的发展速度非常快，每次推出新版本都可能有重大改进，有时还可能不向后兼容。要与时俱进，请务必参阅每版的发行说明。在本章余下的篇幅中，我们将使用 Numba 0.30.1 版。为避免出现错误，请务必安装正确的版本。

本章的完整代码示例可在 notebook Numba.ipynb 中找到。

### 5.1.1 Numba 入门

Numba 很容易上手。作为第一个示例，我们将实现一个函数，它计算一个数组中所有元素的平方和。这个函数的定义如下：

```
def sum_sq(a):
    result = 0
    N = len(a)

    for i in range(N):
        result += a[i]
    return result
```

要让 Numba 对这个函数进行编译，只需将装饰器 `nb.jit` 应用于它。

```
from numba import nb

@nb.jit
def sum_sq(a):
    ...
```

装饰器 `nb.jit` 所做的工作不多，但这个函数首次被调用时，Numba 将检测输入参数（`a`）的类型，并编译出一个性能更高的特殊版本。

要测量 Numba 编译器带来的性能提升，可对原始函数和特殊函数的执行时间进行比较。要访问未经装饰的原始函数，可使用属性 `py_func`。这两个函数的执行时间如下：

```
import numpy as np

x = np.random.rand(10000)
```

```
# 原始函数
%timeit sum_sq.py_func(x)
100 loops, best of 3: 6.11 ms per loop

# Numba 版函数
%timeit sum_sq(x)
100000 loops, best of 3: 11.7 µs per loop
```

从上面的输出可知，Numba 版的速度比 Python 版快一个数量级（前者的执行时间为 11.7 微秒，而后者的执行时间为 6.11 毫秒）。我们还可将这种实现与 NumPy 标准运算符进行比较。

```
%timeit (x**2).sum()
10000 loops, best of 3: 14.8 µs per loop
```

就这个示例而言，Numba 编译得到的函数的速度比 NumPy 向量化运算稍快些。Numba 版本的速度之所以更高，很可能是因为 NumPy 版本在求和前额外分配了数组，而函数 `sum_sq` 在数组中就地执行运算。

由于函数 `sum_sq` 没有使用数组特有的方法，因此也可将这个函数用于包含浮点数的 Python 列表。有趣的是，相比于列表推导，Numba 的速度要快得多。

```
x_list = x.tolist()
%timeit sum_sq(x_list)
1000 loops, best of 3: 199 µs per loop

%timeit sum([x**2 for x in x_list])
1000 loops, best of 3: 1.28 ms per loop
```

鉴于只需应用一个简单的装饰器，就可在计算不同数据类型的平方和时极大地提高速度，因此 Numba 的所作所为就像是在变魔术。在接下来的两小节中，我们将深入探讨 Numba 的工作原理，并对 Numba 编译器的优点和局限性进行评估。

### 5.1.2　类型特殊化

正如你在前面看到的，装饰器 `nb.jit` 在遇到新参数类型后编译函数的特殊版本。为了更好地理解其中的工作原理，可查看 `sum_sq` 示例中经过装饰的函数。

Numba 通过属性 `signatures` 暴露了特殊版本。在函数 `sum_sq` 的定义后面，我们可通过访问 `sum_sq.signatures` 来查看现有的特殊版本，如下所示。

```
sum_sq.signatures
# 输出：
# []
```

如果我们使用特定的参数（如一个 `float64` 数组）调用这个函数，将发现 Numba 动态地编译了一个特殊版本。如果再使用一个 `float32` 数组调用这个函数，将发现列表 `sum_sq.`

signatures 新增了一个元素。

```
x = np.random.rand(1000).astype('float64')
sum_sq(x)
sum_sq.signatures
# 结果:
# [(array(float64, 1d, C),)]

x = np.random.rand(1000).astype('float32')
sum_sq(x)
sum_sq.signatures
# 结果:
# [(array(float64, 1d, C),), (array(float32, 1d, C),)]
```

可显式地针对特定类型来编译这个函数，为此可向函数 nb.jit 传递一个签名。

要传递签名，可使用一个元组，其中包含可接受的类型。Numba 提供了大量的类型，这些类型可在模块 nb.types 和顶级命名空间 nb 中找到。如果要指定特定类型的数组，可将切片运算符（[:]）用于相应的类型。下面的示例演示了如何声明一个将 float64 数组作为唯一参数的函数。

```
@nb.jit((nb.float64[:],))
def sum_sq(a):
```

请注意，显式地声明签名后，就不能使用其他类型了，如下面的示例所示。如果我们试图传递一个 float32 数组（x），Numba 将引发 TypeError 异常。

```
sum_sq(x.astype('float32'))
# TypeError: No matching definition for argument type(s)
array(float32, 1d, C)
```

另一种声明签名的方式是使用指定类型的字符串。例如，要声明一个函数，它将一个 float64 值作为输入，并输出一个 float64 值，可使用字符串 float64(float64)。要声明数组类型，可使用后缀[:]。结合使用这两项规则，可像下面这样声明我们的函数 sum_sq：

```
@nb.jit("float64(float64[:])")
def sum_sq(a):
```

还可传入多个签名，为此可传入一个列表：

```
@nb.jit(["float64(float64[:])",
         "float64(float32[:])"])
def sum_sq(a):
```

### 5.1.3 对象模式和原生模式

前面演示了 Numba 在处理非常简单的函数时的行为。在这种情况下，Numba 的表现非常出色，无论处理的是数组还是列表，性能都得到了极大的提高。

Numba 能够在多大程度上进行优化取决于两个因素：能否准确地推断变量的类型；能否将标准 Python 操作转换为速度更快的、针对特定类型的版本。如果 Numba 能够做到这两点，就能将解释器撇在一边，进而获得类似于使用 Cython 的性能提升。

如果 Numba 无法推导出变量的类型，它依然会对代码进行编译，但在类型无法确定或操作没有得到支持时转而求助于解释器。在 Numba 中，这被称为**对象模式**（object mode），与之相对的是**原生模式**（不需要求助于解释器）。

Numba 提供了一个名为 `inspect_types` 的函数，可帮助你了解类型推断的效果有多好，以及哪些操作被优化了。例如，我们可查看 Numba 为函数 `sum_sq` 所做的类型推断：

```
sum_sq.inspect_types()
```

当你调用这个函数时，Numba 将打印为这个函数的每个版本推断出的类型。输出包含多个部分，其中列出了有关变量及其类型的信息。例如，请看其中的 `N = len(a)` 行。

```
# --- LINE 4 ---
#   a = arg(0, name=a)  :: array(float64, 1d, A)
#   $0.1 = global(len: <built-in function len>) ::
Function(<built-in function len>)
#   $0.3 = call $0.1(a)  :: (array(float64, 1d, A),) -> int64
#   N = $0.3  :: int64

N = len(a)
```

对于每一行，Numba 都打印有关变量、函数和中间结果的详细描述。在上述输出的第 2 行，正确地指出了参数 `a` 的类型是一个 `float64` 数组；而在第 4 行，也正确地指出了函数 `len` 的输入和返回类型，它们分别是 `float64` 数组和 `int64`（可能经过了优化）。

如果你在输出中滚动，将发现所有变量都有明确的类型。由此可以肯定，Numba 能够极其高效地编译这些代码。这种编译被称为**原生模式**。

作为反例，我们来编写一个使用了不支持的操作的函数，看看结果如何。例如，在 0.30.1 版中，Numba 对字符串操作的支持有限。

我们可实现一个拼接一系列字符串的函数，并对其进行编译，如下所示。

```
@nb.jit
def concatenate(strings):
    result = ''
    for s in strings:
        result += s
    return result
```

接下来，我们使用一个字符串列表调用这个函数，并查看类型推导情况。

```
concatenate(['hello', 'world'])
concatenate.signatures
# 输出： [(reflected list(str),)]
concatenate.inspect_types()
```

Numba 的输出表明，生成的特殊版本的签名为 `reflected list(str)`。我们可查看 Numba 是如何对第 3 行代码进行推导的。下面是 `concatenate.inspect_types()` 的输出。

```
# --- LINE 3 ---
#   strings = arg(0, name=strings)  :: pyobject
#   $const0.1 = const(str, )  :: pyobject
#   result = $const0.1  :: pyobject
#   jump 6
#   label 6

result = ''
```

从中可知，这次每个变量（函数参数）的类型都是通用类型 `pyobject`，而不是特定的类型。这意味着如果不求助于 Python 解释器，Numba 将无法对这个操作进行编译。最重要的是，如果我们对原始函数和编译版的执行时间进行测量，将发现编译版的速度比纯粹的 Python 版大约要慢 3 倍。

```
x = ['hello'] * 1000
%timeit concatenate.py_func(x)
10000 loops, best of 3: 111 µs per loop

%timeit concatenate(x)
1000 loops, best of 3: 317 µs per loop
```

这是因为 Numba 编译器不仅无法对代码进行优化，还给函数调用增加了额外的开销。

你可能注意到了，Numba 无声无息地编译代码，即便编译得到的代码的效率更低。其中的主要原因是 Numba 能够高效地编译部分代码，而对于余下的代码，它可求助于 Python 解释器。这种编译策略被称为**对象模式**。

可强制使用原生模式，为此可给装饰器 `nb.jit` 传递选项 `nopython=True`。例如，如果我们在将这个装饰器应用于函数 `concatenate` 时这样做，首次调用这个函数时 Numba 将引发异常。

```
@nb.jit(nopython=True)
def concatenate(strings):
    result = ''
    for s in strings:
        result += s
    return result

concatenate(x)
# 异常：
# TypingError: Failed at nopython (nopython frontend)
```

对调试来说，这项功能很有用，它还可确保所有代码的速度都很快且指定了正确的类型。

## 5.1.4　Numba 和 NumPy

最初开发 Numba 旨在提升使用 NumPy 数组的代码的性能，当前，这个编译器高效地实现了 NumPy 的很多功能。

### 1. Numba 通用函数

通用函数（universal function，ufunc）是 NumPy 中定义的特殊函数，可根据广播规则操作不同长度和形状的数组。Numba 最大的优点之一是实现了快速的 ufunc。

你在第 3 章见过一些 ufunc，例如，np.log 就是一个 ufunc，因为它能够将标量以及长度和形状不同的数组作为输入。另外，接受多个参数的通用函数也根据广播规则进行工作，这样的通用函数包括 np.sum 和 np.difference。

在 NumPy 中，可这样定义通用函数：实现其标量版，并使用函数 np.vectorize 添加广播功能。例如，下面将演示如何编写**康托尔配对函数**（Cantor pairing function）。

配对函数将两个自然数编码成一个自然数，让你能够在这两种表示法之间轻松地转换。可以像下面这样编写康托尔配对函数：

```
import numpy as np

def cantor(a, b):
    return  int(0.5 * (a + b)*(a + b + 1) + b)
```

前面说过，使用纯粹的 Python 时，可使用装饰器 np.vectorized 来创建通用函数。

```
@np.vectorize
def cantor(a, b):
    return  int(0.5 * (a + b)*(a + b + 1) + b)

cantor(np.array([1, 2]), 2)
# 结果：
# array([ 8, 12])
```

如果不考虑便利性，使用纯粹的 Python 来定义通用函数不是很有用，因为这涉及大量存在解释器开销的函数调用。有鉴于此，ufunc 通常是使用 C 或 Cython 实现的，但这些做法都不如 Numba，因为它提供了极大的便利性。

在 Numba 中，要完成定义通用函数所需的转换，只需使用与 np.vectorized 等价的装饰器 nb.vectorize。我们可以比较标准 np.vectorized 版（cantor_py）的速度和使用标准 NumPy 操作实现的版本的速度，如下面的代码所示。

```
# 纯粹的 Python 版本
%timeit cantor_py(x1, x2)
100 loops, best of 3: 6.06 ms per loop
# Numba
```

```
%timeit cantor(x1, x2)
100000 loops, best of 3: 15 µs per loop
# NumPy
%timeit (0.5 * (x1 + x2)*(x1 + x2 + 1) + x2).astype(int)
10000 loops, best of 3: 57.1 µs per loop
```

从中可知，Numba 遥遥领先于其他所有版本！Numba 之所以表现如此出色，是因为这个函数很简单，能够进行类型推导。

> 通用函数的另一个优点是，可并行地执行，因为它们依赖于各个值。Numba 提供了对这种函数进行并行化的简单方式：向装饰器 `nb.vectorize` 传递关键字参数 `target="cpu"`或 `target="gpu"`。

### 2. 泛化通用函数

通用函数的一个局限性是其定义必须针对标量值。泛化通用函数（generalized universal function，gufunc）是对接受数组的过程的通用函数扩展。

一个典型的示例是矩阵乘法。在 NumPy 中，要执行矩阵乘法运算，可使用函数 `np.matmul`，它接受两个二维数组并返回一个二维数组。下面是一个使用 `np.matmul` 的示例。

```
a = np.random.rand(3, 3)
b = np.random.rand(3, 3)
c = np.matmul(a, b)
c.shape
# 结果：
# (3, 3)
```



前一小节说过，ufunc 将操作广播到整个**标量**数组，因此一种自然而然的泛化是广播到**数组**的数组。例如，有两个由 3×3 矩阵组成的数组，我们希望 `np.matmul` 能够计算它们的乘积。在下面的示例中，有两个数组，它们分别包含 10 个形状为`(3, 3)`的矩阵。如果将它们传递给 `np.matmul`，将计算相应矩阵的乘积，得到一个新数组，其中包含 10 个结果（而每个结果也都是形状为`(3, 3)`的矩阵）。

```
a = np.random.rand(10, 3, 3)
b = np.random.rand(10, 3, 3)
c = np.matmul(a, b)
c.shape
# 输出
# (10, 3, 3)
```

广播规则的工作原理与此类似。例如，如果有一个由`(3, 3)`矩阵组成的数组（其形状为`(10, 3, 3)`），可使用 `np.matmul` 来对其中的每个元素与一个`(3, 3)`的矩阵执行矩阵乘法运算。根据广播规则，将通过重复这个`(3, 3)`矩阵，得到一个形状为`(10, 3, 3)`的矩阵：

```
a = np.random.rand(10, 3, 3)
b = np.random.rand(3, 3) # Broadcasted to shape (10, 3, 3)
c = np.matmul(a, b)
```

```
c.shape
# 结果：
# (10, 3, 3)
```

Numba 提供了装饰器 nb.guvectorize，以支持高效的泛化通用函数实现。例如，我们将实现一个泛化通用函数，它计算两个数组的欧几里得距离。要创建 gufunc，必须定义一个这样的函数：它将数组作为输入，并返回一个包含计算结果的数组。

装饰器 nb.guvectorize 接受两个参数。

- 输入和输出的类型：两个作为输入的一维数组和一个作为输出的标量。
- 表示输入和输出长度的布局字符串；在这里，输入是两个长度相同的数组（用两个 n 表示），输出是一个标量。

下面的示例演示了如何使用装饰器 nb.guvectorize 来实现函数 euclidean。

```
@nb.guvectorize(['float64[:], float64[:], float64[:]'], '(n), (n) -
> ()')
def euclidean(a, b, out):
    N = a.shape[0]
    out[0] = 0.0
    for i in range(N):
        out[0] += (a[i] - b[i])**2
```

这里有几个非常重要的地方需要说一说。我们将输入（a 和 b）的类型声明为 float64[:]，因为它们是一维数组。但输出参数呢？前面不是说它是一个标量吗？没错，是标量，但 Numba 将标量视为长度为 1 的数组，因此这里将它声明为 float64[:]。

同样，布局字符串指出函数接受两个长度为 n 的数组，而其输出是一个标量，用一对空的括号（()）表示。然而，传递的输出将是一个长度为 1 的数组。

另外请注意，这个函数什么都不返回，因此所有输出都必须写入到数组 out 中。

> **TIP** 前述布局字符串中的字母 n 是随意选择的，你也可使用字母 k 或你喜欢的其他字母。另外，如果要组合长度不同的数组，可使用诸如(n, m)这样的字符串。

这个全新的函数 euclidean 可用于不同形状的数组，如下面的示例所示。

```
a = np.random.rand(2)
b = np.random.rand(2)
c = euclidean(a, b) # 形状：(1,)

a = np.random.rand(10, 2)
b = np.random.rand(10, 2)
c = euclidean(a, b) # 形状：(10,)

a = np.random.rand(10, 2)
b = np.random.rand(2)
c = euclidean(a, b) # 形状：(10,)
```

与标准 NumPy 相比，函数 euclidean 的速度怎样呢？在下面的代码中，我们对向量化的 NumPy 版本和刚定义的函数 euclidean 进行了基准测试。

```
a = np.random.rand(10000, 2)
b = np.random.rand(10000, 2)

%timeit ((a - b)**2).sum(axis=1)
1000 loops, best of 3: 288 µs per loop

%timeit euclidean(a, b)
10000 loops, best of 3: 35.6 µs per loop
```

同样，Numba 版本遥遥领先于 NumPy 版本！

### 5.1.5 JIT 类

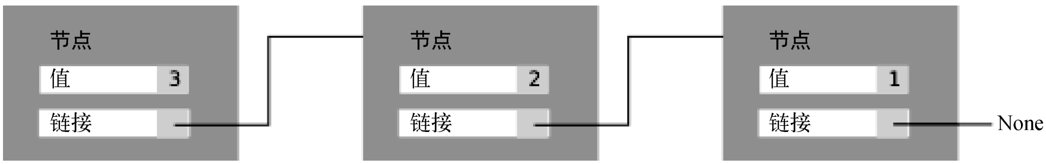
当前，Numba 不支持对泛型 Python 对象进行优化。然而，这种局限性对数值计算代码的影响不大，因为这种代码通常只涉及数组和数学运算。

尽管如此，对有些数据结构来说，使用对象实现起来要自然得多，因此 Numba 支持定义类，而这些类可编译成快速的原生代码。

别忘了，这是一种最新推出的功能，但很有用，因为它允许我们扩展 Numba，以支持使用数组不容易实现的数据结构。



作为例子，我们将演示如何使用 JIT 类来实现简单的链表。要实现链表，可定义一个 Node 类，这个类包含两个字段：一个值以及一个到下一个节点的链接。如下图所示，每个节点都链接到下一个节点并包含一个值，而最后一个节点包含一个断开的链接，我们将这个链接的值设置为 None。




在 Python 中，我们可以这样定义 Node 类：

```
class Node:
    def __init__(self, value):
        self.next = None
        self.value = value
```

为管理一系列 Node 实例，可创建另一个类——LinkedList。这个类跟踪链表的开头（在上图中，这对应于值为 3 的节点）。要在链表开头插入一个节点，只需新建一个 Node 实例，并

将其链接到当前的链表头。

在下面的代码中，我们为 LinkedList 定义了初始化函数和方法 push_back，其中方法 push_back 使用前面所说的策略在链表开头插入一个节点。

```
class LinkedList:

    def __init__(self):
        self.head = None

    def push_front(self, value):
        if self.head == None:
            self.head = Node(value)
        else:
            # 替换链表头
            new_head = Node(value)
            new_head.next = self.head
            self.head = new_head
```

为方便调试，我们还可实现方法 LinkedList.show，它遍历并打印链表中的每个节点。这个方法的代码如下所示：

```
def show(self):
    node = self.head
    while node is not None:
        print(node.value)
        node = node.next
```

现在可对 LinkedList 进行测试，看看它的行为是否正确。为此，可创建一个空链表，添加几个节点，并打印链表的内容。请注意，由于我们在链表开头压入节点，因此最后插入的节点将最先打印。

```
lst = LinkedList()
lst.push_front(1)
lst.push_front(2)
lst.push_front(3)
lst.show()
# 输出：
# 3
# 2
# 1
```

最后，我们可实现一个函数——sum_list，它返回链表中所有节点值之和。我们将测量这个方法的 Numba 版本和纯粹的 Python 版本在执行时间上的差别。

```
@nb.jit
def sum_list(lst):
    result = 0
    node = lst.head
    while node is not None:
    result += node.value
```

```
        node = node.next
    return result
```

如果测量原始版 sum_list 和 nb.jit 版的执行时间，将发现没有多大的差别。这是因为 Numba 无法推断类的类型。

```
lst = LinkedList()
[lst.push_front(i) for i in range(10000)]

%timeit sum_list.py_func(lst)
1000 loops, best of 3: 2.36 ms per loop

%timeit sum_list(lst)
100 loops, best of 3: 1.75 ms per loop
```

为改进 sum_list 的性能，可使用装饰器 nb.jitclass 来编译 Node 和 LinkedList 类。

装饰器 nb.jitclass 接受一个参数，其中包含被装饰类的属性的类型。在 Node 类中，属性 value 的类型为 int64，而属性 next 的类型为 Node。装饰器 nb.jitclass 还会编译类的所有方法。深入探讨代码前，还有两点需要说明。

首先，必须先声明属性，再定义类，但如何声明还未定义的类型呢？Numba 提供了函数 nb.deferred_type()，可用来完成这项任务。

其次，属性 next 可以是 None，也可以是一个 Node 实例。这被称为可选类型，而 Numba 提供了实用工具 nb.optional，让你得以指出变量的取值可能为 None。



下面的代码示例演示了如何将装饰器应用于 Node 类。如你所见，预先使用 nb.deferred_type()声明了 node_type。属性是在一个列表中声明的，该列表中的每个元素都包含属性名和类型（另外请注意，其中还使用了 nb.optional）。声明 Node 类后，必须声明延迟的类型（deferred type）。

```
node_type = nb.deferred_type()

node_spec = [
    ('next', nb.optional(node_type)),
    ('value', nb.int64)
]

@nb.jitclass(node_spec)
class Node:
    # Node 类的类体没变

node_type.define(Node.class_type.instance_type)
```

LinkedList 类很容易编译，只需定义属性 head，并应用装饰器 nb.jitclass 即可，如下所示。

```
ll_spec = [
    ('head', nb.optional(Node.class_type.instance_type))
```

```
]

@nb.jitclass(ll_spec)
class LinkedList:
    # LinkedList 类的类体没变
```

现在可以测量传入 JIT 类 `LinkedList` 的实例时，函数 `sum_list` 的执行时间了。

```
lst = LinkedList()
[lst.push_front(i) for i in range(10000)]

%timeit sum_list(lst)
1000 loops, best of 3: 345 µs per loop

%timeit sum_list.py_func(lst)
100 loops, best of 3: 3.36 ms per loop
```

有趣的是，在编译型函数中使用 JIT 类时，性能相比于纯粹的 Python 版本有极大的提升。然而，在原始函数 `sum_list.py_func` 中使用 JIT 类时，性能反而降低了。因此，请务必只在编译型函数中使用 JIT 类！

### 5.1.6 Numba 的局限性

在有些情况下，Numba 无法正确地推断出变量的类型，进而拒绝编译。在下面的示例中，我们定义了一个函数，它接受一个嵌套的整数列表，并返回每个子列表中所有元素之和。在这个示例中，Numba 将引发 `ValueError` 异常，并拒绝编译。

```
a = [[0, 1, 2],
     [3, 4],
     [5, 6, 7, 8]]

@nb.jit
def sum_sublists(a):
    result = []
    for sublist in a:
        result.append(sum(sublist))
    return result

sum_sublists(a)
# ValueError: cannot compute fingerprint of empty list
```

这些代码存在的问题是，Numba 无法确定列表 `result` 的类型，因此以失败告终。要修复这种问题，一种办法是将列表 `result` 初始化为包含一个元素，以帮助编译器确定该列表的类型，并在最后将这个元素删除。

```
@nb.jit
def sum_sublists(a):
    result = [0]
    for sublist in a:
        result.append(sum(sublist))
    return result[1:]
```

在 Numba 编译器中还没有实现的功能包括函数和类定义、列表、集推导和字典推导、生成器、with 语句以及 try except 块。但请注意，其中许多功能未来都可能得到支持。

## 5.2 PyPy 项目

PyPy 是一个野心勃勃的项目，旨在改善 Python 解释器的性能，这是通过在运行阶段自动编译速度缓慢的代码实现的。

PyPy 是使用特殊语言 RPython（而不是 C 语言）编写的，让开发人员能够快速而可靠地实现高级功能和改进。RPython 的意思是“受限的 Python”（restricted Python），因为它只实现了 Python 语言中用于开发编译器的那部分。

当前，PyPy 5.6 版支持大量的 Python 功能，完全可用于编写众多不同的应用程序。

PyPy 采用一种非常巧妙的策略来编译代码，这种策略被称为**跟踪式 JIT 编译**（tracing JIT compilation）。一开始，PyPy 像通常那样使用解释器调用来执行代码，然后开始剖析代码，找出耗时最多的循环。找出这些循环后，PyPy 编译器观察（**跟踪**）操作，并编译出经过优化且不需要解释器的版本。

有了优化版代码后，PyPy 就能够以比解释型版本快得多的速度，运行原本速度缓慢的循环。

这种策略与 Numba 的做法形成了鲜明的对比。在 Numba 中，编译单元为方法和函数，而 PyPy 只专注于速度缓慢的循环。总体而言，这两个项目的关注点也有天壤之别：Numba 只能优化执行数值计算的代码，且需要做大量的准备工作，而 PyPy 致力于取代 CPython 解释器。

接下来将演示如何使用 PyPy 来执行粒子模拟器，并进行基准测试。

### 5.2.1 安装 PyPy

PyPy 是以预先编译好的二进制文件分发的，这个二进制文件可从 http://pypy.org/download.html 下载。当前，PyPy 支持 Python 2.7（在 PyPy 5.6 中为 beta 支持）和 Python 3.3（在 PyPy 5.5 中为 alpha 支持）。本章将演示如何在 Python 2.7 中使用 PyPy。

下载并解压缩 PyPy 后，就可在解压缩到的文件夹中找到这个解释器，它位于该文件夹下的目录 bin/pypy 中。你可使用下面的命令初始化一个新的虚拟环境，以便在其中安装其他的包。

```
$ /path/to/bin/pypy -m ensurepip
$ /path/to/bin/pypy -m pip install virtualenv
$ /path/to/bin/virtualenv my-pypy-env
```

要激活这个环境，可使用如下命令：

```
$ source my-pypy-env/bin/activate
```



5

现在可以核实 Python 链接到 PyPy 可执行文件了，为此可使用命令 `python -V`。你还可以安装其他一些可能需要用到的包。PyPy 5.6 版对使用 Python C API 的软件（其中最著名的是 numpy 和 matplotlib 包）提供了有限的支持。我们可像通常那样安装这些包：

```
(my-pypy-env) $ pip install numpy matplotlib
```

TIP 在有些平台上，numpy 和 matplotlib 安装起来可能比较棘手。你可不安装这些包，并在后面将运行的脚本中删除导入它们的语句。

### 5.2.2　在 PyPy 中运行粒子模拟器

安装 PyPy 后，就可以运行粒子模拟器了。首先，我们将使用标准 Python 解释器来执行第 1 章的粒子模拟器，并测量其执行时间。如果虚拟环境还处于活动状态，可使用命令 deactivate 来退出。为核实 Python 解释器为标准解释器，可使用命令 `python -V`。

```
(my-pypy-env) $ deactivate
$ python -V
Python 3.5.2 :: Continuum Analytics, Inc.
```

现在可在命令行界面中使用 timeit 来测量代码的执行时间了。

```
$ python -m timeit --setup "from simul import benchmark" "benchmark()"
10 loops, best of 3: 886 msec per loop
```

我们可重新激活虚拟环境，并在 PyPy 中运行这些代码。在 Ubuntu 中，导入模块 matplotlib.pyplot 时可能会遇到麻烦。为修复此问题，可尝试执行下面的 export 命令，也可将导入 matplotlib 的语句从 simul.py 中删除。

```
$ export MPLBACKEND='agg'
```

现在使用 PyPy 来执行这些代码并测量执行时间。

```
$ source my-pypy-env/bin/activate
Python 2.7.12 (aff251e54385, Nov 09 2016, 18:02:49)
[PyPy 5.6.0 with GCC 4.8.2]

(my-pypy-env) $ python -m timeit --setup "from simul import benchmark"
"benchmark()"
WARNING: timeit is a very unreliable tool. use perf or something else for
real measurements
10 loops, average of 7: 106 +- 0.383 msec per loop (using standard
deviation)
```

注意，性能得到了极大的提升，速度快了 8 倍多！然而，PyPy 警告我们，模块 timeit 可能不可靠。为核实时间测量结果，可像 PyPy 建议的那样使用模块 perf。

```
(my-pypy-env) $ pip install perf
(my-pypy-env) $ python -m perf timeit --setup 'from simul import benchmark'
```

```
'benchmark()'
.......
Median +- std dev: 97.8 ms +- 2.3 ms
```

## 5.3 其他有趣的项目

多年来，有很多项目试图通过多种不同的策略来改善 Python 的性能，但遗憾的是其中很多都以失败告终。当前，幸存的项目只有几个，它们有望提高 Python 的速度。

Numba 和 PyPy 都是成熟的项目，多年来一直在不断改善。它们不断地增加功能，是 Python 未来的希望所在。

Nuitka 是 Kay Hayen 开发的一个程序，它将 Python 代码编译成 C 代码。当前（0.5.x 版），它与 Python 语言的兼容性很高，能够生成高效的代码，这些代码在性能上的提升比 CPython 还高些。

相比于 Cython，Nuitka 有天壤之别，因为它专注于与 Python 语言兼容，没有通过添加额外的结构来扩展 Python。

Pyston 是 Dropbox 开发的一款新解释器，用于支持 JIT 编译器。它与 PyPy 完全不同：不使用跟踪式 JIT，而是使用每次一个方法的 JIT（就像 Numba 所做的那样）。与 Numba 一样，Numba 也建立在 LLVM 编译器基础设施之上。

Pyston 还处于早期开发阶段（alpha 阶段），只支持 Python 2.7。基准测试表明，其速度比 CPython 快，但比 PyPy 慢。虽然如此，它还是一个值得跟踪的项目，因为它添加了新功能，并且提高了兼容性。



## 5.4 小结

Numba 是一个在运行阶段编译 Python 函数的快速专用版本的工具。本章介绍了如何使用 Numba 来编译函数以及如何查看并分析它们，还介绍了如何实现快速的 NumPy 通用函数，这些函数在很多数值计算应用程序中都很有用。最后，我们使用装饰器 `nb.jitclass` 实现了一些比较复杂的数据结构。

诸如 PyPy 等工具让你能够原样运行 Python 程序，并极大地提高速度。我们演示了如何安装 PyPy，并使用它来运行了粒子模拟器，以看看性能改善情况。

我们还简要地介绍了当前的 Python 编译器生态系统，并对这些编译器做了比较。

下一章将介绍并发性和异步编程。对于那些花大量时间等待网络和磁盘资源的应用程序，使用这些技术可改善其设计和响应速度。

第 6 章

# 实现并发性

本书前面探索了如何测量程序的性能,以及如何通过巧妙的算法和高效的机器码来减少 CPU 执行的操作数，进而改善程序的性能。在有些程序中，大部分时间都花在等待速度比 CPU 慢得多的资源（如永久性存储和网络资源）上，本章将把注意力转向这样的程序。

异步编程是一种编程范式，可帮助你处理速度缓慢且不可预测的资源（如用户），它被广泛用于打造响应迅速的服务和用户界面。本章将介绍如何在 Python 中使用协程和响应式编程等技术来进行异步编程。

本章介绍如下主题：

- 存储器层次结构；
- 回调函数；
- future；
- 事件循环；
- 使用 `asyncio` 编写协程；
- 将同步代码转换为异步代码；
- 使用 RxPy 进行响应式编程；
- 使用被观察者（observable）；
- 使用 RxPy 打造内存监视器。

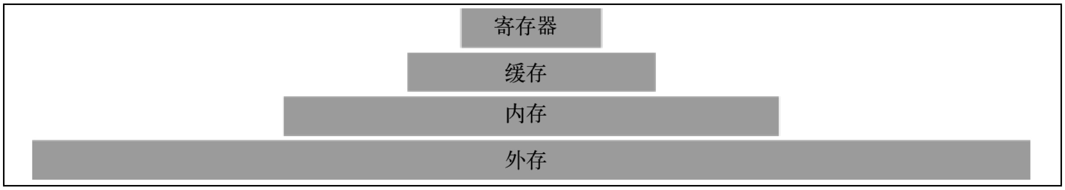
## 6.1 异步编程

异步编程是一种处理缓慢且不可预测资源的方式。异步程序能够高效地同时处理多种资源，而不是坐在那里等待资源可用。异步编程可能很难，因为这需要处理外部请求，而这些外部请求的到达顺序可能不可预测，处理它们所需的时间可能不是固定的，还可能意外地失败。本节将介绍异步编程的主要概念和术语，以及工作原理。

### 6.1.1　等待 I/O

现代计算机利用各种不同的存储器来存储数据和执行操作。通常，计算机中包含昂贵但运行速度快的内存，还有价格便宜但运行速度缓慢的存储器，后者通常用来存储大量的数据。

存储器的层次结构如下图所示。




在存储器层次结构的顶端是 CPU 寄存器，它们集成在 CPU 中，用于存储和执行机器指令。访问寄存器中的数据所需的时间通常为一个时钟周期，这意味着如果 CPU 的频率为 3GHz，访问 CPU 寄存器中一个元素所需的时间大约为 0.3 纳秒。

**寄存器**下面那层是 CPU 缓存。缓存有多级，也被集成到处理器中。缓存的速度比寄存器慢些，但在一个数量级内。

存储器层次结构中的接下来一层是主存（**内存**），它能够存储的数据比缓存多得多，但速度更慢。从内存中获取一个元素所需的时间可能是几百个时钟周期。

最底层是永久性存储，如旋转磁盘（HDD）和**固态硬盘**（SSD）。这些设备能够存储的数据最多，但速度比主存差几个数量级。为寻找并获取一个元素，HDD 可能需要几毫秒，而 SSD 的速度则快得多，需要的时间不到 1 毫秒。

为了让你对各种存储器的相对速度有清楚的认识，可以打个比方：如果 CPU 的时钟周期约为 1 秒，访问寄存器将相当于从桌子上取支铅笔，访问缓存相当于从书架上取本书，访问内存相当于将衣物在干洗机上放好（比访问缓存慢 20 倍）。永久性存储的速度相差很大：从 SSD 中获取一个元素相当于 4 天的旅行，而从 HDD 中获取一个元素需要 6 个月！如果要通过网络访问资源，需要的时间将更长。

前面的示例清楚地说明，相比于 CPU，访问永久性存储和其他 I/O 设备中数据的速度要慢得多。因此处理这些资源时，不让 CPU 漫无目的地等待至关重要。为确保这一点，可精心地设计软件，使其能够同时管理多个请求。

### 6.1.2　并发

并发是一种实现系统同时处理多个请求的方式，其基本理念是在等待资源期间可着手处理其他的资源。并发的工作原理是：将任务划分成可不按顺序执行的子任务，这样就能同时处理多个

子任务，而无须等到前面的子任务完成。

在第一个示例中，我们将介绍如何实现对缓慢的网络资源的并发访问。假设有一个 Web 服务，它获取数字的平方；另外，假设从请求该 Web 服务到获得响应的时间大约为 1 秒。我们可实现函数 network_request，它接受一个数字，并返回一个字典，其中包含有关计算是否成功的信息以及计算结果。为模拟该 Web 服务，可使用函数 time.sleep，如下所示。

```
import time

def network_request(number):
    time.sleep(1.0)
    return {"success": True, "result": number ** 2}
```

另外，我们还编写其他一些代码，它们发出请求、核实请求成功并打印结果。在下面的代码中，我们定义了函数 fetch_square，并使用它来计算数字 2 的平方（而它通过调用 network_request 计算这个数的平方）。

```
def fetch_square(number):
    response = network_request(number)
    if response["success"]:
        print("Result is: {}".format(response["result"]))

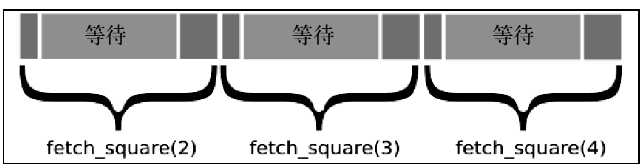
fetch_square(2)
# 输出：
# 结果： 4
```

由于网络速度缓慢，从网络获取一个数字需要 1 秒。如果要计算多个数字的平方，该怎么办呢？我们可多次调用 fetch_square，其中每个调用都将在前一个调用结束后发起网络请求。

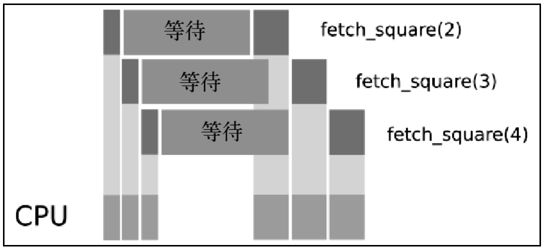
```
fetch_square(2)
fetch_square(3)
fetch_square(4)
# 输出：
# 结果： 4
# 结果： 9
# 结果： 16
```

上述代码需要 3 秒才能执行完毕，但这样的结果不是最佳的。没必要等待前一个请求结束，因为从技术上说，我们可以同时提交多个请求，再等待它们的结果。

在下图中，用方框表示了这三个任务，其中 CPU 为处理并提交请求而花费的时间为深灰色，而等待时间为浅灰色。从中可知，大部分时间都花在等待资源上，而在等待期间，机器在那里干坐着，什么都没做。



理想情况下，应在等待已提交的任务结束时开始另一个新任务。在下图中，提交 fetch_square(2)中的请求后，就可开始为 fetch_square(3)做准备了。这样可减少 CPU 等待时间，并在有了结果后立即着手处理。




之所以能够采取这种策略，是因为这三个请求是完全独立的，无须等到前一个任务完成后再着手处理下一个任务。另外，单个 CPU 就能得心应手地处理这种情形。虽然将工作分配给多个 CPU 去完成可进一步提高执行速度，但如果相对于处理时间而言等待时间很长，这样做带来的速度提升将很有限。

要实现并发，必须以不同的方式思考和编写代码。下一节将介绍实现健壮的并发应用程序的技巧和最佳实践。

### 6.1.3 回调函数

前面介绍的代码阻塞程序的执行，直到资源可用，其中负责等待的调用是 time.sleep。为了让代码立即着手处理其他任务，需要想办法避免阻塞程序流程，让程序的其他部分能够继续完成其他任务。

6

为此，最简单的办法之一是使用回调函数，其策略与我们叫出租车时很像。

假设你在饭店喝了几杯酒，而外面下着雨，你不想去坐公交车，于是决定叫辆出租车，并让司机到达后给你打电话。这样你将在接到电话后再出饭店，避免在雨中等待。

在这种情况下，你叫了辆出租车（速度较慢的资源），但不在饭店外等待出租车到来，而是向司机提供电话号码和说明（回调函数），以便等出租车到了后再出来乘车回家。

下面来演示如何在代码中使用这种机制。我们将对阻塞代码 time.sleep 与非阻塞代码 threading.Timer 进行比较。

在这个示例中，我们将编写一个函数——wait_and_print，它将程序执行流程阻塞一秒钟，再打印一条消息。

```
def wait_and_print(msg):
    time.sleep(1.0)
    print(msg)
```

要以非阻塞方式编写这个函数，可使用 threading.Timer 类。我们可初始化一个 threading.Timer 实例：传递要等待的时长以及一个回调函数。**回调函数**不过是一个将在定时器到期后被调用的函数。请注意，我们还必须调用方法 Timer.start 来激活定时器。

```
import threading

def wait_and_print_async(msg):
    def callback():
        print(msg)

    timer = threading.Timer(1.0, callback)
    timer.start()
```

函数 wait_and_print_async 的一个重要特征是，其中的所有语句都不会阻塞程序的执行流程。

> TIP　threading.Timer 是如何做到在等待的同时不阻塞的呢？threading.Timer 使用的策略是启动一个新线程，该线程能够并行地执行代码。如果这不好理解，也不用担心，本书后面将详细介绍线程和并行编程。

这种注册回调函数，以便在特定事件发生时执行它的方式称为**好莱坞原则**，这是因为在好莱坞，演员试镜后可能被告知“不要给我们打电话，我们会给你打电话”，这意味着他们不会立即告诉你是否选定了你，但如果选定了你，会给你电话的。

为突出阻塞版 wait_and_print 和非阻塞版的差别，可比较这两个版本的执行情况。在下面的输出注释中，<等待……>表示等待过程。

```
# 同步的
wait_and_print("First call")
wait_and_print("Second call")
print("After call")
# 输出：
# <等待……>
# First call
# <等待……>
# Second call
# After call
# 异步的
wait_and_print_async("First call async")
wait_and_print_async("Second call async")
print("After submission")
# 输出：
# After submission
# <等待……>
# First call
# Second call
```

同步版本的行为与以前很像：代码等待 1 秒钟，打印 First call，再等待 1 秒钟，然后打印消息 Second call 和 After call。

在异步版本中，wait_and_print_async **提交**（而不是**执行**）这些调用并立即接着往前走。从输出可知，立即打印了消息 After submission，这说明这种机制发挥了作用。

知道这些后，就可以探索更复杂些的情形了：使用回调函数重写函数 network_request。在下面的代码中，我们定义了函数 network_request_async。相比于阻塞版，函数 network_request_async 最大的不同在于它**什么都没返回**，这是因为在 network_request_async 被调用时，我们只提交请求，而结果要等到请求完成后才能得到。

既然什么都不能返回，如何传递请求的结果呢？我们将结果作为参数传递给回调函数 on_done，而不是返回它。

在这个函数余下的代码中，向 timer.Timer 类提交了一个回调函数（timer_done），它将在准备就绪后调用 on_done。

```
def network_request_async(number, on_done):

    def timer_done():
        on_done({"success": True,
                 "result": number ** 2})
    timer = threading.Timer(1.0, timer_done)
    timer.start()
```

network_request_async 的用法与 timer.Timer 很像，只需传递一个要计算其平方的数字，以及在结果准备就绪后将收到它的回调函数，如下面的代码所示。

```
def on_done(result):
    print(result)

network_request_async(2, on_done)
```

现在，如果你提交多个网络请求，将发现调用将并发地执行，而不会阻塞代码。

```
network_request_async(2, on_done)
network_request_async(3, on_done)
network_request_async(4, on_done)
print("After submission")
```

要在 fetch_square 中使用 network_request_async，需要对其进行修改，以使用异步结构。在下面的代码中，我们修改了 fetch_square——定义回调函数 on_done 并将其传递给 network_request_async。

```
def fetch_square(number):
    def on_done(response):
        if response["success"]:
```

6

```
        print("Result is: {}".format(response["result"]))

    network_request_async(number, on_done)
```

你可能注意到了，相比于同步代码，异步代码要复杂得多。这是因为每次需要获取结果时，都必须编写并传递一个回调函数，这导致代码一层套一层，变得难以理解。

### 6.1.4　future

future 是一种更便利的模式，可用来跟踪异步调用的结果。在前面的代码中，没有返回结果，而是接受一个回调函数，并在结果就绪后将其传递给这个回调函数。有趣的是，到目前为止，没有跟踪资源状态的简单途径。

future 是一种抽象，可帮助我们跟踪请求的资源并等到它可用。在 Python 中，`concurrent.futures.Future` 类提供了一种 future 实现。要创建这个类的实例，可调用其构造函数且不提供任何参数。

```
fut = Future()
# 结果：
# <Future at 0x7f03e41599e8 state=pending>
```

future 表示一个还不可用的值，其字符串表示指出了结果的当前状态（这里为 pending，即还未确定）。要让结果可用，可使用方法 `Future.set_result`。

```
fut.set_result("Hello")
# 结果：
# <Future at 0x7f03e41599e8 state=finished returned str>

fut.result()
# 结果：
# "Hello"
```

如你所见，设置结果后，`Future` 将指出任务结束了，此时可使用方法 `Future.result` 来访问结果。还可给 future 指定一个回调函数，这样一旦结果可用，就将执行这个回调函数。要指定回调函数，只需向方法 `Future.add_done_callback` 传递一个函数即可。这样任务结束后，指定的函数将被调用，并将 `Future` 实例作为第一个参数。在指定的回调函数中，可使用方法 `Future.result()`来访问结果。

```
    fut = Future()
    fut.add_done_callback(lambda future: print(future.result(),
flush=True))
    fut.set_result("Hello")
    # 输出：
    # Hello
```

为了让你掌握如何在实际工作中使用 future，我们将修改函数 `network_request_async`，在其中转而使用 future。这里的理念是，不再什么都不返回，而是返回一个 `Future`，用于跟踪

结果。这里需要注意两点。

- 不再接受回调函数 on_done，因为可在以后使用方法 Future.add_done_callback 来关联到回调函数。另外，将通用方法 Future.set_result 作为回调函数传递给 threading.Timer。
- 这次可以返回一个值，因此代码与前一节介绍的阻塞版本更像。

```
from concurrent.futures import Future

def network_request_async(number):
    future = Future()
    result = {"success": True, "result": number ** 2}
    timer = threading.Timer(1.0, lambda: future.set_result(result))
    timer.start()
    return future

fut = network_request_async(2)
```

> 在这些示例中，我们直接实例化并管理 future，但在实际应用程序中，future 是由框架处理的。

如果你执行上述代码，什么都不会发生，因为这些代码只是创建并返回一个 Future 实例。要进一步操作 future 的结果，需要使用方法 Future.add_done_callback。在下面的代码中，我们修改函数 fetch_square 以使用 future。

```
def fetch_square(number):
    fut = network_request_async(number)

    def on_done_future(future):
        response = future.result()
        if response["success"]:
            print("Result is: {}".format(response["result"]))

    fut.add_done_callback(on_done_future)
```

这些代码依然与回调版本很像。future 提供了另一种使用回调函数的方式，而且更方便些。使用 future 也更好，因为它们能够跟踪资源状态、撤销已调度的任务以及以更自然的方式处理异常。

### 6.1.5 事件循环

前面使用了操作系统线程来实现并行，但在很多异步框架中，并发任务之间的协调工作是由**事件循环**管理的。

事件循环背后的理念是，不断地监视各种资源（如网络链接和数据库查询）的状态，并在事件发生（如资源准备就绪或定时器到期）时执行相应的回调函数。

**为何不坚持使用线程呢？**

在有些情况下，事件循环是更佳的选择，因为每个执行单元都不会与其他执行单元同时运行，这简化了共享变量、数据结构和资源的处理工作。有关并行执行及其缺点的详细信息，请参阅下一章。

在本节的第一个示例中，我们将实现 `threading.Timer` 的非线程版本。我们可定义一个 `Timer` 类，它接受超时时间，并实现方法 `Timer.done`（这个方法在定时器到期时返回 `True`）。

```
class Timer:

    def __init__(self, timeout):
        self.timeout = timeout
        self.start = time.time()

    def done(self):
        return time.time() - self.start > self.timeout
```

为判断定时器是否已到期，可编写一个循环，它不断调用方法 `Timer.done` 来检查定时器的状态。定时器到期后，我们可打印一条消息，并退出循环。

```
timer = Timer(1.0)

while True:
    if timer.done():
        print("Timer is done!")
        break
```

通过以这样的方式实现定时器，可确保执行流程绝不会被阻塞，因此从原则上说，可在这个 `while` 循环中执行其他操作。

通过使用循环不断轮询来等待事件发生，这通常被称为**忙等待**（busy-waiting）。

理想情况下，应指定一个在定时器到期时执行的自定义函数，就像 `threading.Timer` 中那样。为此，可实现方法 `Timer.on_timer_done`，它接受一个要在定时器到期时执行的回调函数。

```
class Timer:
    # ……以前的代码
    def on_timer_done(self, callback):
        self.callback = callback
```

请注意，`on_timer_done` 只是存储了一个指向回调函数的引用，负责监视事件并执行回调函数的是循环。下面来演示这一点。在这里，不再在循环中使用函数 `print`，而是在合适的情况下调用 `timer.callback`。

```
timer = Timer(1.0)
timer.on_timer_done(lambda: print("Timer is done!"))
```

```
while True:
    if timer.done():
        timer.callback()
        break
```

如你所见，一个异步框架的雏形已经形成。在循环外面，我们只定义了定时器和回调函数，监视定时器和执行回调函数的工作由循环负责。我们可进一步扩展这些代码，以支持多个定时器。

为实现多个定时器，一种自然而然的方式是将多个 `Timer` 实例添加到一个列表中，并修改循环，使其定期地检查所有的定时器，并在必要时调用回调函数。在下面的代码中，我们定义了两个定时器，并将每个定时器都关联到一个回调函数。这些定时器被添加到一个列表（`timers`）中。事件循环不断地监视这个列表，一旦有定时器到期，就执行相应的回调函数，并将该定时器从列表中删除。

```
timers = []

timer1 = Timer(1.0)
timer1.on_timer_done(lambda: print("First timer is done!"))

timer2 = Timer(2.0)
timer2.on_timer_done(lambda: print("Second timer is done!"))

timers.append(timer1)
timers.append(timer2)

while True:
    for timer in timers:
        if timer.done():
            timer.callback()
            timers.remove(timer)
    # 如果列表中没有任何定时器，就退出循环
    if len(timers) == 0:
        break
```

6

事件循环的主要局限性在于**绝不能使用阻塞调用**，因为执行流程是由不断运行的循环管理的。可以想见，如果在循环中使用了阻塞语句（如 `time.sleep`），事件监视和回调函数分派将停止，直到阻塞调用完成。

为避免这种情况发生，我们不使用阻塞调用（如 `time.sleep`），而是让事件循环负责检测资源是否已就绪，并在资源就绪后调用回调函数。通过避免阻塞执行流程，事件循环可同时监视多项资源。

> 事件通知通常是通过操作系统调用（如 Unix 工具 `select`）实现的，操作系统调用会在事件就绪后恢复程序执行，而不是忙等待。

Python 标准库包含一个基于事件循环的并发框架——`asyncio`，它使用起来很方便，这将在下一节介绍。

## 6.2 asyncio 框架

至此，你应对并发的工作原理以及如何使用回调函数和 future 有了深入的认识，可以接着学习如何使用 asyncio 包了（从 Python 3.4 起，Python 标准库就包含这个包）。我们还将探索全新的 async/await 语法，这是一种非常自然的异步编程方式。

在本节的第一个示例中，我们将介绍如何使用 asyncio 获取并执行一个简单的回调函数。要获取 asyncio 循环，可调用函数 asyncio.get_event_loop()。要调度回调函数，可使用 loop.call_later，它接受以秒为单位的延迟和一个回调函数。你还可使用方法 loop.stop 来停止循环并退出程序。要开始处理已调度的调用，必须启动循环，为此可使用 loop.run_forever。下面的示例演示了如何使用这些基本方法，它调度了一个打印消息并停止循环的回调函数。

```
import asyncio

loop = asyncio.get_event_loop()

def callback():
    print("Hello, asyncio")
    loop.stop()

loop.call_later(1.0, callback)
loop.run_forever()
```

### 6.2.1 协程

使用回调函数的一个主要问题是，必须将程序划分成在特定事件发生时将被调用的小型函数。正如你在本章前面看到的，回调函数很容易变得非常烦琐。

协程是另一种（可能更自然的）将程序划分成小块的方式，让程序员能够编写看起来像同步代码但将异步执行的代码。可将协程视为可停止和恢复执行的函数。一个简单的协程示例是生成器。

在 Python 中，要定义生成器，可在函数中使用 yield 语句。在下面的示例中，我们实现了函数 range_generator，它生成并返回值 0 到 n。我们还添加了一条 print 语句，以显示生成器的内部状态。

```
def range_generator(n):
    i = 0
    while i < n:
        print("Generating value {}".format(i))
        yield i
        i += 1
```

当你调用函数 range_generator 时，其中的代码并不会立即执行。请注意，下面的代码执行时，什么都没有打印，而只是返回一个 generator 对象。

```
generator = range_generator(3)
generator
# 结果：
# <generator object range_generator at 0x7f03e418ba40>
```

要从生成器中取值，必须使用函数 next：

```
next(generator)
# 输出：
# Generating value 0

next(generator)
# 输出：
# Generating value 1
```

请注意，每当我们调用 next 时，都将运行代码，直到遇到 yield 语句。要让生成器接着往下执行，必须再次调用 next。你可将 yield 语句视为一个断点，可执行到这里停止，还可从这里开始继续执行（同时保持生成器的内部状态不变）。可在事件循环中利用这种停止和继续执行功能来实现并发。

还可使用 yield 语句将值**注入**生成器（而不是从中**提取**值）。在下面的示例中，我们声明了函数 parrot，它重复我们发送的每条消息。要让生成器接收值，可将 yield 赋给一个变量（在这里，使用的是语句 message = yield）。要将值插入生成器，可使用方法 send。在 Python 中，能够接收值的生成器称为**基于生成器的协程**。

```
def parrot():
    while True:
        message = yield
        print("Parrot says: {}".format(message))

generator = parrot()
generator.send(None)
generator.send("Hello")
generator.send("World")
```

请注意，开始发送消息前，必须调用 generator.send(None)，这旨在将函数执行到第一条 yield 语句。另外，注意函数 parrot 中有一个无限循环，如果不使用生成器，这个循环将没完没了地执行下去！

基于前面的介绍，你完全可以想见，事件循环可让多个生成器逐步推进，而不阻塞整个程序的执行流程。你还可以想见，生成器可仅在相关资源就绪时才往前推进，从而不需要使用回调函数。

在 asyncio 中，可使用 yield 语句来实现协程，但从 3.5 版起，Python 支持使用更直观的

语法来定义功能强大的协程。

要使用 asyncio 来定义协程，可使用语句 async def：

```
async def hello():
    print("Hello, async!")

coro = hello()
coro
# 输出：
# <coroutine object hello at 0x7f314846bd58>
```

如你所见，调用函数 hello 时，没有立即执行其代码，而是返回了一个 coroutine 对象。asyncio 协程不支持 next，但可在 asyncio 事件循环中轻松地运行它们，为此只需使用方法 run_until_complete 即可。

```
loop = asyncio.get_event_loop()
loop.run_until_complete(coro)
```

使用 async def 语句定义的协程也称**原生协程**。

模块 asyncio 提供了资源（被称为 awaitable），你可在协程中使用 await 语法来请求它们。例如，如果你要等待一段时间后再执行语句，可使用函数 asyncio.sleep。

```
async def wait_and_print(msg):
    await asyncio.sleep(1)
    print("Message: ", msg)

loop.run_until_complete(wait_and_print("Hello"))
```

这样编写出来的代码漂亮而整洁。我们根本没有使用丑陋的回调函数，就编写出了功能完美的异步代码！

你可能注意到了，await 给事件循环提供了一个断点，因此在等待资源期间，事件循环可继续管理其他协程。

锦上添花的是，协程也是 awaitable，因此可使用 await 语句将协程异步地串接起来。在下面的示例中，我们重写了本章前面定义的函数 network_request——将 time.sleep 替换为 asyncio.sleep。

```
async def network_request(number):
    await asyncio.sleep(1.0)
    return {"success": True, "result": number ** 2}
```

接下来，可重新实现 fetch_square。如你所见，可直接等待（await）network_request，而不需要额外的 future 或回调函数。

```
async def fetch_square(number):
    response = await network_request(number)
    if response["success"]:
        print("Result is: {}".format(response["result"]))
```

可使用 loop.run_until_complete 分别运行协程：

```
loop.run_until_complete(fetch_square(2))
loop.run_until_complete(fetch_square(3))
loop.run_until_complete(fetch_square(4))
```

对测试和调试来说，使用 run_until_complete 来运行任务很好，但在大多数情况下，程序都将首先执行 loop.run_forever，因此需要在循环已经在运行的情况下提交任务。

asyncio 提供了函数 ensure_future，可用来调度协程（和 future）。要使用 ensure_future，只需将要调度的协程传递给它即可。下面的代码调度多个 fetch_square 调用，这些调用将并发地执行。

```
asyncio.ensure_future(fetch_square(2))
asyncio.ensure_future(fetch_square(3))
asyncio.ensure_future(fetch_square(4))

loop.run_forever()
# 要停止循环，可按 Ctrl-C
```

另外，向它传入一个协程时，函数 asyncio.ensure_future 将返回一个 Task 实例（Task 是 Future 的子类），这让我们既能使用 await 语法，又能利用 future 的资源跟踪功能。

### 6.2.2 将阻塞代码转换为非阻塞代码

虽然 asyncio 支持以异步方式连接到资源，但在有些情况下必须使用阻塞调用，例如，在第三方 API（如数据库访问库）只暴露了阻塞调用时，或执行长时间运行的计算时。在这一节中，我们将介绍如何处理阻塞 API，使其与 asyncio 兼容。

对于阻塞代码，一种有效的处理策略是在一个独立的线程中运行它们。线程是在**操作系统**（OS）层级实现的，允许阻塞代码并行地执行。Python 提供了接口 Executor，它设计用于在独立的线程中运行任务，并使用 future 来监视任务的进度。

要初始化 ThreadPoolExecutor，必须先从模块 concurrent.futures 导入它。这种执行器会生成一系列线程（被称为工作线程），这些线程将等待执行抛给它们的任何任务。函数被提交后，执行器将负责将执行它的工作分派给空闲的工作线程，并跟踪结果。要指定线程数量，可使用参数 max_workers。

请注意，任务结束后，执行器不会销毁相应的线程，这样可降低创建和销毁线程的开销。

在下面的示例中，我们创建了一个包含三个工作线程的 ThreadPoolExecutor，并提交了

函数 wait_and_return，这个函数将程序执行流程阻塞 1 秒钟，并返回一个消息字符串。然后，我们使用方法 submit 调用这个函数，将其交给线程去执行。

```
from concurrent.futures import ThreadPoolExecutor
executor = ThreadPoolExecutor(max_workers=3)

def wait_and_return(msg):
    time.sleep(1)
    return msg

executor.submit(wait_and_return, "Hello. executor")
# 结果：
# <Future at 0x7ff616ff6748 state=running>
```

方法 executor.submit 立即调度这个函数，并返回一个 future。在 asyncio 中，可使用方法 loop.run_in_executor 来管理任务的执行，这个方法的工作原理与 executor.submit 很像。

```
fut = loop.run_in_executor(executor, wait_and_return, "Hello, asyncio
executor")
# <Future pending ...more info...>
```

方法 run_in_executor 也返回一个 asyncio.Future 实例，你可在其他代码中等待这个实例；主要差别在于，这个 future 仅在我们启动循环后才会运行。要运行它并获取响应，可使用 loop.run_until_complete。

```
loop.run_until_complete(fut)
# 结果：
# 'Hello, executor'
```

作为一个实例，我们可使用这种技巧同时获取多个网页。为此，我们导入流行的（阻塞）库 requests，并在执行器中运行函数 requests.get。

```
import requests

async def fetch_urls(urls):
    responses = []
    for url in urls:
        responses.append(await loop.run_in_executor
                         (executor, requests.get, url))
    return responses

loop.run_until_complete(fetch_ruls(['http://www.google.com',
                                    'http://www.example.com',
                                    'http://www.facebook.com']))
# 结果：
# []
```

这个版本的 fetch_urls 不会阻塞执行，允许 asyncio 中的其他协程运行，但它并不是最优的，因为它不会并行地获取 URL。要并行地获取 URL，可使用 asyncio.ensure_future，

也可使用便利函数 asyncio.gather（它一次性提交所有的协程并收集到来的结果）。下面演示了 asyncio.gather 用法。

```
def fetch_urls(urls):
    return asyncio.gather(*[loop.run_in_executor
                            (executor, requests.get, url)
                            for url in urls])
```

> TIP 使用这个方法可并行地获取的 URL 数量取决于有多少个工作线程。为避免这种限制，应使用非阻塞原生库，如 aiohttp。

## 6.3 响应式编程

响应式编程是一种编程范式，旨在打造更出色的并发系统。响应式应用程序符合响应式宣言（reactive manifesto）规定的要求。

- **响应速度快**：系统迅速地响应用户。
- **伸缩性高**：系统能够处理不同水平的负载，并能够适应更严苛的需求。
- **富有弹性**：系统能够妥善地应对故障，这是通过模块化以及避免单点故障实现的。
- **消息驱动**：系统不应阻塞，并利用事件和消息。消息驱动的应用程序有助于满足前述所有需求。

如你所见，响应式系统目标远大，但响应式编程到底是如何工作的呢？本节将以 RxPy 库为例介绍响应式编程的原理。

> ReactiveX 是一个项目，实现了用于众多语言的响应式编程工具，而 RxPy 是其中的一个库。



### 6.3.1 被观察者

顾名思义，响应式编程的主要理念是对事件做出**响应**。前一节介绍了一些使用回调函数实现这种理念的示例：注册回调函数，使其在事件发生时立即执行。

响应式编程扩展了这种理念，它将事件视为数据流。为证明这一点，我们来看看 RxPy 中一些这样的流。可基于迭代器来创建数据流，为此可使用工厂方法 Observable.from_iterable，如下所示。

```
from rx import Observable
obs = Observable.from_iterable(range(4))
```

要接收来自 obs 的数据，可使用方法 Observable.subscribe，这样将对数据源发射（emit）的每个值执行传入的函数。

```
obs.subscribe(print)
# 输出:
# 0
# 1
# 2
# 3
```

你可能注意到了，被观察者是有序的元素集合，就像列表（推而广之是像迭代器）一样。这并非巧合。

> 术语 observable（被观察者）由 observer（观察者）和 iterable（可迭代对象）组合而成。**观察者**是一个对象，在其观察的变量发生变化时做出反应，而**可迭代对象**能够生成并跟踪迭代器。

在 Python 中，迭代器是定义了方法__next__的对象，你可通过调用 next 来提取其元素。迭代器通常是通过对集合执行 iter 来生成的。生成迭代器后，就可使用 next 或 for 循环来提取元素。提取迭代器中的一个元素后，就不能回过头去再提取。下面从一个列表创建一个迭代器，以演示迭代器的用法。

```
collection = list([1, 2, 3, 4, 5])
iterator = iter(collection)

print("Next")
print(next(iterator))
print(next(iterator))

print("For loop")
for i in iterator:
    print(i)

# 输出:
# Next
# 1
# 2
# For loop
# 3
# 4
# 5
```

从这个示例可知，每当我们调用 next 或进行迭代时，迭代器都生成一个值并前进一步。从某种意义上说，这是从迭代器中**提取**结果。

> 迭代器看起来很像生成器，但更通用。在 Python 中，生成器是由使用 yield 表达式的函数返回的。你知道，生成器支持 next，因此是一种特殊的迭代器。

至此，你明白迭代器和被观察者的异同了。被观察者准备就绪后向我们**推送**一个数据流，它还能够在出现错误或没有更多数据时告诉我们。实际上，可使用方法 Observable.subscribe 注册其他回调函数。在下面的示例中，我们创建了一个被观察者，并使用参数 on_next 和

on_completed 注册了两个回调函数，这两个回调函数将分别在下一项数据可用以及没有更多数据时被调用。

```
obs = Observable.from_iter(range(4))
obs.subscribe(on_next=lambda x: print(on_next="Next item: {}"),
              on_completed=lambda: print("No more data"))
# 输出：
# Next element: 0
# Next element: 1
# Next element: 2
# Next element: 3
# No more data
```

这种与迭代器的类似性很重要，因为我们可以使用处理迭代器的方法来处理事件流。

RxPy 提供了可用来创建、变换和过滤被观察者以及对其进行编组的运算符。响应式编程的威力在于，这些操作返回其他被观察者，因此可将它们串接和组合在一起。为了让你体会这一点，下面来演示 take 运算符的用法。

给定一个被观察者，take 返回一个新的被观察者，但这个被观察者只提供前 n 个元素。这个运算符的用法非常简单。

```
obs = Observable.from_iterable(range(100000))
obs2 = obs.take(4)

obs2.subscribe(print)
# 输出：
# 0
# 1
# 2
# 3
```

6

RxPy 实现了丰富而多样的运算符，你可使用它们来创建复杂的应用程序。

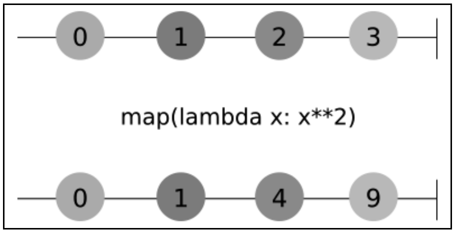
### 6.3.2 很有用的运算符

本节将探索以某种方式对源被观察者的元素进行变换的运算符。在这个运算符家族中，最重要的成员是 map，它对源被观察者的元素执行指定的函数，再将结果发射出去。例如，可使用 map 来计算一系列数字的平方。

```
(Observable.from_iterable(range(4))
           .map(lambda x: x**2)
           .subscribe(print))
# 输出：
# 0
# 1
# 4
# 9
```

可使用弹珠图（marble diagram）来表示运算符，这可帮助我们更好地理解运算符的工作原理，元素可能在特定时间区域内发射出去时尤其如此。在弹珠图中，实线表示数据流（这里是一个被观察者），圆形（或其他形状）表示被观察者发射的值，符号 X 表示错误，而垂直线表示数据流的终点。

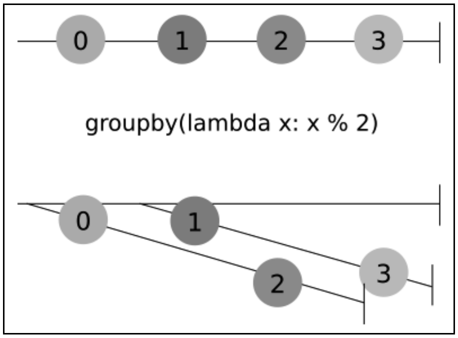
下面是前述 map 运算符的弹珠图。




源被观察者位于弹珠图顶部，中间是变换，而底部是生成的被观察者。

另一个变换运算符 group_by，它根据键将元素编组。运算符 group_by 接受一个函数，这个函数根据提供给它的元素提取键，并为每个键生成一个新的被观察者，其中包含与该键相关联的元素。

使用弹珠图可将 group_by 操作更清晰地呈现出来。在下图中，group_by 发射两个被观察者。另外，元素被发射后，就被动态地编组。




为了深入理解 group_by 的工作原理，来看一个简单的示例。假设我们要根据数字是奇数还是偶数对其进行分组，为此可使用 group_by，并将表达式 lambda x:x % 2 作为键函数传递给它。这个键函数在数字为偶数时返回 0，在数字为奇数时返回 1。

```
obs = (Observable.from_range(range(4))
             .group_by(lambda x: x % 2))
```

现在，如果我们订阅并打印 obs 的内容，将打印两个被观察者。

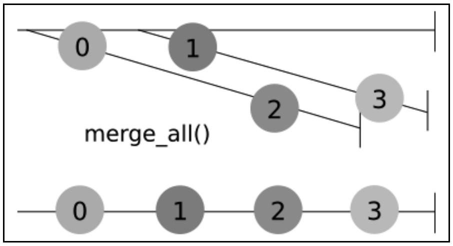
```
obs.subscribe(print)
# <rx.linq.groupedobservable.GroupedObservable object at 0x7f0fba51f9e8>
# <rx.linq.groupedobservable.GroupedObservable object at 0x7f0fba51fa58>
```

要获悉分组的键，可使用属性 key。要提取所有的偶数，可使用 take 运算符获取第一个被观察者（其键值为 0）并订阅它，如下面的代码所示。

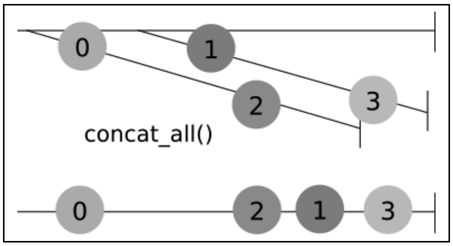
```
obs.subscribe(lambda x: print("group key: ", x.key))
# 输出：
# group key: 0
# group key: 1
obs.take(1).subscribe(lambda x: x.subscribe(print))
# 输出：
# 0
# 2
```

通过使用 group_by，我们引入了发射其他被观察者的被观察者。在响应式编程中，这是一种常见的模式。还有一些函数能够合并不同的被观察者。

在合并被观察者方面，两个很有用的工具是 merge_all 和 concat_all。merge_all 接受多个被观察者，并生成一个被观察者，其中包含接受的被观察者中的所有元素，而这些元素的排列顺序与发射顺序相同。可使用弹珠图来更好地说明这一点。




merge_all 类似于 concat_all，但后者返回一个新的被观察者，这个被观察者先发射第一个被观察者中的元素，再发射第二个被观察者中的元素，以此类推。concat_all 的弹珠图如下。




为了演示这两个运算符的用法，我们将它们应用于 group_by 返回的被观察者的被观察者。merge_all 将按元素的初始顺序返回它们（别忘了 group_by 将元素分两组发射）。

```
obs.merge_all().subscribe(print)
# 输出
# 0
# 1
# 2
# 3
```

而 concat_all 先返回偶数元素，再返回奇数元素，因为它先发射第一个被观察者中的元素，发射完毕后再发射第二个被观察者中的元素，下面的代码演示了这一点。在这个示例中，我们还使用了函数 make_replay。为何需要使用它呢？因为使用完“偶数”流后，第二个流中的元素已发射完毕，concat_all 无法使用它们。等你阅读 6.3.3 节后，将会深刻地理解这一点。

```
def make_replay(a):
    result = a.replay(None)
    result.connect()
    return result

obs.map(make_replay).concat_all().subscribe(print)
# 输出
# 0
# 2
# 1
# 3
```

这次先打印的是偶数，打印完偶数后才打印奇数。

TIP　RxPy 还提供了操作 merge 和 concat，你可使用它们来合并独立的被观察者。

### 6.3.3　hot 被观察者和 cold 被观察者

前一节介绍了如何使用方法 Observable.from_iterable 来创建被观察者。RxPy 提供了很多其他的工具，可用来创建更有趣的事件源。

Observable.interval 接受一个以毫秒为单位的时间段（参数 period），并创建一个每隔指定时间就发射一个值的被观察者。下面的代码行创建一个被观察者（obs），这个被观察者从零开始每秒发射一个数字。我们使用了运算符 take 对这个定时器进行限制，使其只触发 4 个事件。

```
obs = Observable.interval(1000)
obs.take(4).subscribe(print)
# 输出：
# 0
# 1
# 2
# 3
```

对于 Observable.interval，非常重要的一点是，它生成的定时器在订阅后才会启动。为证明这一点，可打印索引以及相对于定义定时器时的延迟（这是使用 time.time()计算得到的），如下所示。

```
import time

start = time.time()
obs = Observable.interval(1000).map(lambda a:
                                       (a, time.time() - start))

# 我们等 2 秒后再订阅
time.sleep(2)
obs.take(4).subscribe(print)
# 输出：
# (0, 3.003735303878784)
# (1, 4.004871129989624)
# (2, 5.005947589874268)
# (3, 6.00749135017395)
```

如你所见，第一个元素（对应于索引 0）是在 3 秒后生成的，这意味着这个定时器在我们调用方法 subscribe(print)后才启动。

Observable.interval 生成的被观察者被称为**惰性**（lazy）的，因为它们等到被请求后才开始生成值（可将这样的被观察者视为自动售卖机，仅当顾客按下按钮后才吐出商品）。用 Rx 的话说，这种被观察者是 **cold** 的。cold 被观察者的一个特征是，如果将其关联到两个订阅者，定时器将启动多次，这一点在下面的示例中非常明显。在这里，我们在第一次订阅 0.5 秒后再次订阅。如你所见，两次订阅的输出时间是不同的：

```
start = time.time()
obs = Observable.interval(1000).map(lambda a:
                                       (a, time.time() - start))
# 我们等 2 秒后再订阅
time.sleep(2)
obs.take(4).subscribe(lambda x: print("First subscriber:
                                         {}".format(x)))
time.sleep(0.5)
obs.take(4).subscribe(lambda x: print("Second subscriber:
                                         {}".format(x)))
# 输出：
# First subscriber: (0, 3.0036110877990723)
# Second subscriber: (0, 3.5052847862243652)
# First subscriber: (1, 4.004414081573486)
# Second subscriber: (1, 4.506155252456665)
# First subscriber: (2, 5.005316972732544)
# Second subscriber: (2, 5.506817102432251)
# First subscriber: (3, 6.0062034130096436)
# Second subscriber: (3, 6.508296489715576)
```

在有些情况下，这种行为可能是你不想要的，因为你希望多个订阅者订阅同一个数据源。要让被观察者只生成一份数据，可使用 publish 来延迟数据生成，确保所有订阅者得到的数据相同。

publish 将被观察者转换为 ConnectableObservable，后者不会立即推送数据，而是等到你调用方法 connect 后才推送数据。下面的代码演示了 publish 和 connect 的用法。

```
start = time.time()
obs = Observable.interval(1000).map(lambda a: (a, time.time() -
start)).publish()
obs.take(4).subscribe(lambda x: print("First subscriber:
                                        {}".format(x)))
obs.connect() # 现在才开始生成数据

time.sleep(2)
obs.take(4).subscribe(lambda x: print("Second subscriber:
                                        {}".format(x)))
# 输出：
# First subscriber: (0, 1.0016899108886719)
# First subscriber: (1, 2.0027990341186523)
# First subscriber: (2, 3.003532648086548)
# Second subscriber: (2, 3.003532648086548)
# First subscriber: (3, 4.004265308380127)
# Second subscriber: (3, 4.004265308380127)
# Second subscriber: (4, 5.005320310592651)
# Second subscriber: (5, 6.005795240402222)
```

在这个示例中，我们先调用方法 publish，再关联第一个订阅者，然后调用方法 connect。调用方法 connect 后，定时器将开始生成数据。第二个订阅者后加入，事实上，它没有收到前两条消息，而是从第三条消息开始接收。注意，这次两个订阅者共享同一份数据。这种不为订阅者分别生成数据的数据源被称为是 hot 的。

你还可使用类似于 publish 的方法 replay，它为每个订阅者从头开始生成数据，如下面的示例所示。除了将 publish 替换成了 replay 外，这个示例与前一个示例相同。

```
import time

start = time.time()
obs = Observable.interval(1000).map(lambda a: (a, time.time() -
start)).replay(None)
obs.take(4).subscribe(lambda x: print("First subscriber:
                                        {}".format(x)))
obs.connect()

time.sleep(2)
obs.take(4).subscribe(lambda x: print("Second subscriber:
                                        {}".format(x)))

First subscriber: (0, 1.0008857250213623)
```

```
First subscriber: (1, 2.0019824504852295)
Second subscriber: (0, 1.0008857250213623)
Second subscriber: (1, 2.0019824504852295)
First subscriber: (2, 3.0030810832977295)
Second subscriber: (2, 3.0030810832977295)
First subscriber: (3, 4.004604816436768)
Second subscriber: (3, 4.004604816436768)
```

如你所见，这次虽然第二个订阅者后加入，但它依然获得了之前发射的所有数据。

另一种创建 hot 被观察者的方式是使用 `Subject` 类。这个类很有趣，既能接收数据，又能推送数据，因此可用来手动将数据**推送**给被观察者。`Subject` 使用起来非常简单。在下面的代码中，我们创建了一个 `Subject` 并订阅它，然后使用方法 `on_next` 向它推送值。向它推送值后，订阅者就立即被调用。

```
s = Subject()
s.subscribe(lambda a: print("Subject emitted value: {}".format(x))
s.on_next(1)
# Subject emitted value: 1
s.on_next(2)
# Subject emitted value: 2
```

请注意，`Subject` 也是 hot 被观察者。

## 6.3.4　打造 CPU 监视器

掌握主要的响应式编程概念后，就可实现一个示例应用程序了。在这一节中，我们将实现一个监视器，它向我们提供有关 CPU 使用率的实时信息，并能够检测出使用率峰值。



这个 CPU 监视器的完整代码可在文件 cpu_monitor.py 中找到。

首先，我们来实现一个数据源。我们将使用模块 `psutil`，它提供了一个名为 `psutil.cpu_percent` 的函数，这个函数以百分比的方式返回最近的 CPU 使用率（且不阻塞）。

```
import psutil
psutil.cpu_percent()
# 结果： 9.7
```

由于我们要开发的是监视器，因此要每隔一段时间采集一次这种信息。为此，可像前一节那样使用熟悉的 `Observable.interval` 和 `map`。另外，我们还要让这个被观察者是 hot 的，因为对这个应用程序而言，所有订阅者收到的应该是同一份数据。为让 `Observable.interval` 是 hot 的，可使用方法 `publish` 和 `connect`。创建被观察者 `cpu_data` 的代码如下：

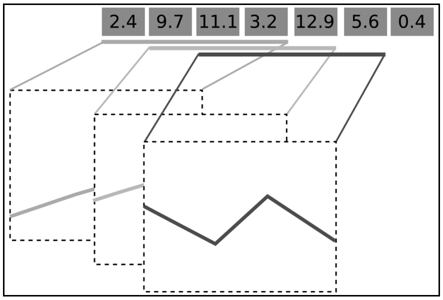
```
cpu_data = (Observable
            .interval(100) # 每隔 100 毫秒
```

```
            .map(lambda x: psutil.cpu_percent())
            .publish())
cpu_data.connect() # 开始生成数据
```

为测试这个监视器，可打印 4 个元素。

```
cpu_data.take(4).subscribe(print)
# 输出：
# 12.5
# 5.6
# 4.5
# 9.6
```

主数据源就绪后，就可使用 matplotlib 实现监视器可视化了。这里的理念是，创建一个图表，其中包含的数据量是固定的；有新数据到来后，就在图表中包含这个最新的数据，并将最旧的数据删除。这通常被称为**移动窗口**，可通过图示更深入地认识它。在下图中，数据流 cpu_data 由一系列数字表示。获得前 4 个数字后，就绘制第一个图表，而每当有新数字到来后，就将窗口移动一个位置，并更新图表。




为实现这种算法，可编写一个函数——monitor_cpu，它将创建并更新绘图窗口。这个函数所做的工作如下。

- 初始化一个空图表，并设置正确的坐标轴范围。
- 对被观察者 cpu_data 进行变换，以返回一个随数据移动的窗口。为此可使用运算符 buffer_with_count，它将窗口中的点数（npoints）作为参数，并移动一个位置。
- 订阅这个新的数据流，并使用到来的数据更新图表。

这个函数的完整代码如下所示。如你所见，代码非常紧凑。请花点时间运行这个函数，并尝试使用不同的参数。

```
import numpy as np
from matplotlib import pyplot as plt

def monitor_cpu(npoints):
    lines, = plt.plot([], [])
    plt.xlim(0, npoints)
```

```
plt.ylim(0, 100) # 0%到100%

cpu_data_window = cpu_data.buffer_with_count(npoints, 1)

def update_plot(cpu_readings):
    lines.set_xdata(np.arange(npoints))
    lines.set_ydata(np.array(cpu_readings))
    plt.draw()

cpu_data_window.subscribe(update_plot)

plt.show()
```

你可能想添加的一个功能是，在 CPU 使用率持续高企一定时间后触发警报，因为这可能昭示着计算机正在做艰难的处理。为此，可结合使用 `buffer_with_count` 和 `map`。我们可获取 CPU 使用率数据流和一个窗口，再在函数 `map` 中检查是否所有 CPU 使用率都超过 20%（在四核 CPU 中，这相当于单个处理器的使用率为 100%）。如果窗口中所有点的值都超过 20%，我们就在图表窗口中显示一条警告消息。

这个新的被观察者的实现如下。它在 CPU 使用率持续高企时发射 `True`，否则发射 `False`。

```
alertpoints = 4
high_cpu = (cpu_data
            .buffer_with_count(alertpoints, 1)
            .map(lambda readings: all(r > 20 for r in readings)))
```

被观察者 `high_cpu` 准备就绪后，就可创建 `matplotlib` 标签，并订阅这个被观察者以更新标签了。

```
label = plt.text(1, 1, "normal")
def update_warning(is_high):
    if is_high:
        label.set_text("high")
    else:
        label.set_text("normal")
high_cpu.subscribe(update_warning)
```

6

## 6.4 小结

在需要处理速度缓慢且无法预测的资源（如 I/O 设备和网络）时，异步编程很有用。本章探索了重要的并发和异步编程概念，并介绍了如何使用 `asyncio` 和 RxPy 库来编写并发代码。

处理多个彼此关联的资源时，`asyncio` 协程是极佳的选择，因为它们巧妙地避开了回调函数，从而极大地简化了代码逻辑。在这些情形下，响应式编程也是不错的选择，但它真正擅长的是处理实时应用程序和用户界面中常见的数据流。

接下来的两章将介绍并行编程，以及如何利用多核和多台计算机极大地改善性能。

# 第 7 章 并行处理

通过使用多核进行并行处理，无须速度更快的处理器，就可让程序在给定时间内执行更多的计算。这里的主要理念是将问题划分成独立的子单元，并使用多个内核并行地处理这些子单元。

对处理大型问题来说，并行处理必不可少。公司每天都生成海量的数据，需要存储在多台计算机中并进行分析。科学家和工程师在超级计算机上运行并行代码来模拟庞大的系统。

并行处理让你能够利用多核 CPU 以及擅长处理高度并行问题的 GPU。本章介绍如下主题：

- 并行处理原理；
- 使用 Python 库 `multiprocessing` 并行地处理简单问题；
- 使用简单接口 `ProcessPoolExecutor`；
- 通过 Cython 和 OpenMP 使用多线程进行并行编程；
- 使用 Theano 和 Tensorflow 自动实现并行性；
- 使用 Theano、Tensorflow 和 Numba 在 GPU 中执行代码。

## 7.1 并行编程简介

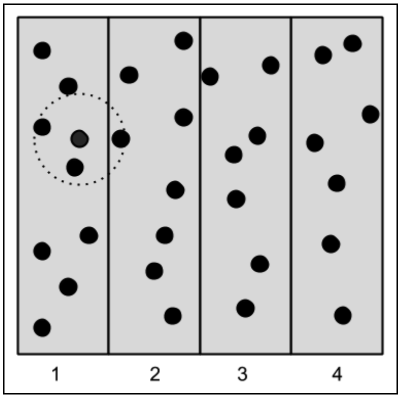
要让程序并行地运行，必须将问题划分为可彼此独立（或几乎独立）运行的子单元。

如果一个问题的各个子单元是完全彼此独立的，这个问题就是**高度并行**的（embarrassingly parallel）。对数组的各个元素分别执行的操作就是一个典型的例子——这种操作只需知道当前处理的元素。另一个例子是本书的粒子模拟器：由于彼此不影响，每个粒子都是独立地运动的。对于高度并行的问题，其解决方案很容易实现，在并行架构上的性能也非常高。

有些问题可划分为不同的子单元，但不同子单元涉及的计算需要共享数据。在这种情况下，解决方案实现起来不那么容易，还可能因为通信开销带来性能问题。

我们将通过一个示例来演示这一点。假设有个粒子模拟器，其中的粒子在距离位于特定范围内时会彼此吸引，如下图所示。为并行地处理这个问题，我们将模拟箱划分成区域，其中每个区域都由一个不同的处理器来负责处理。如果我们每次计算一步，有些粒子将与邻接区域内的粒子

交互。为完成下一次迭代，相邻区域之间必须通告粒子的新位置。



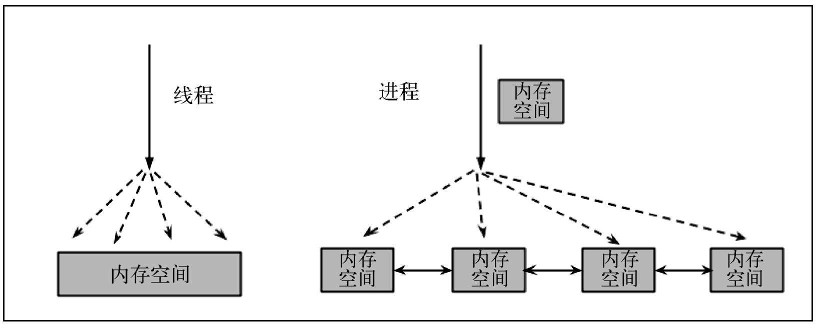

进程间通信的开销非常高，可能严重影响并行程序的性能。在并行程序中，处理数据通信的方式主要有两种：

- **共享内存**
- **分布式内存**

在共享内存中，各个子单元可访问相同的内存空间。这种方法的优点在于，你无须显式地处理通信，因为只需读写共享内存就够了。然而，多个进程试图同时访问并修改相同的内存单元时，将出现问题。因此，必须使用同步技术避免这样的冲突。

在分布式内存模型中，每个进程都与其他进程完全分开，并有自己的内存空间。在这种情况下，必须显式地处理进程之间的通信。与共享内存相比，通信开销通常更高，因为数据可能穿过网络接口。

以共享内存方式实现并行的一种常见方式是使用**线程**。线程是源自进程的独立子任务，并共享内存等资源。下图进一步说明了这个概念。线程生成多个执行上下文并共享内存空间，而进程提供多个执行上下文，有自己的内存空间，因此必须显式地处理通信。



Python 能够生成并处理线程，但使用线程不能改善性能。由于 Python 解释器的设计，每次只能运行一个 Python 指令，这种机制称为**全局解释器锁**（GIL）。每当线程执行 Python 语句时，都获取一个锁，执行完毕后，再释放这个锁。由于每次只有一个线程能够获得这个锁，因此一个线程获得这个锁后，其他线程就不能执行 Python 语句。

虽然 GIL 导致 Python 指令无法并行执行，但在可释放这个锁的情况下（如在耗时的 I/O 操作或 C 语言扩展中），依然可使用线程来实现并发。

> 为何不将 GIL 删除呢？过去几年，有过很多这样的尝试，其中包括最近的 GIL 切除术（gilectomy）实验。首先，要删除 GIL 并不那么容易，必须修改大部分 Python 数据结构。另外，细粒度的锁定可能代价高昂，还可能导致单线程程序的性能急剧下降。虽然如此，有些 Python 实现就没有使用 GIL，其中最著名的是 Jython 和 IronPython。

通过使用进程而不是线程，可完全避开 GIL。进程不共享内存区域，而且是彼此独立的——每个进程都有自己的解释器。进程有一些缺点：启动新进程通常比启动新线程慢；它们消耗的内存更多；进程间通信的速度可能很慢。另一方面，进程也非常灵活，分布在多台计算机中时可伸缩性更佳。

## 图形处理单元

图形处理单元是特殊的处理器，是为运行计算机图形学应用程序而设计的。这些应用程序通常需要处理 3D 场景的几何结构，并将像素数组输出到屏幕上。GPU 执行的操作包括浮点数数组和矩阵运算。

GPU 就是为高效地运行与图形相关的操作而设计的，这是通过采用高度并行的体系结构来实现的。相比于 CPU，GPU 包含的小型处理单元要多得多（数千个）。GPU 以每秒 60 帧的速度生成数据，这比时钟速度更高的 CPU 的典型响应速度慢得多。

GPU 专门用于执行浮点数运算，其体系结构与标准 CPU 有天壤之别。因此，要编译供 GPU 运行的程序，必须使用特殊的编程平台，如 CUDA 和 OpenCL。

**统一计算设备体系结构**（compute unified device architecture，CUDA）是一种 NVIDIA 专用的技术，提供了可在其他语言中访问的 API。CUDA 提供了工具 NVCC，可用来编译使用 CUDA C 语言（类似于 C）编写的 GPU 程序；它还提供了大量的库，这些库实现了高度优化的数学例程。

OpenCL 是一种开放技术，使用它编写的并行程序可针对各种目标平台（不同厂商生产的 CPU 和 GPU）进行编译，因此对非 NVIDIA 设备来说，使用 OpenCL 是个不错的选择。

GPU 编程好像很神奇，但你千万不要因此而抛弃 CPU。GPU 编程很棘手，而且仅在特定情况下你才能受益于 GPU 体系结构。程序员必须明白将数据写入内存以及从内存读取数据的成本，还必须知道如何实现算法以充分发挥 GPU 体系结构的作用。

一般而言，GPU 可极大地提高单位时间内可执行的操作数（即**吞吐量**），但它们需要更多的时间来准备要处理的数据。相反，CPU 从头开始生成单个结果的速度要快得多（这被称为**延时**）。

对于合适的问题，使用 GPU 可极大地提高速度（高达 10~100 倍），因此，在改善数值密集型应用程序的性能方面，GPU 提供了极其廉价的解决方案（要实现同样的速度提升，需要数百个 CPU）。7.4 节将演示如何在 GPU 上执行一些算法。

## 7.2　使用多个进程

标准模块 `multiprocessing` 可用来生成多个进程，以快速并行化简单任务，同时避免 GIL 问题。这个模块的接口使用起来很容易，其中包含多个处理任务提交和同步的实用工具。

### 7.2.1　Process 和 Pool 类

要创建独立运行的进程，可从 `multiprocessing.Process` 派生出子类。可通过扩展方法 `__init__` 来初始化资源，还可通过实现方法 `Process.run` 来编写将在子进程中执行的代码。在下面的代码中，我们定义了一个 `Process` 类，它等待 1 秒钟再打印分配给自己的 `id`。

```
import multiprocessing
import time

class Process(multiprocessing.Process):
    def __init__(self, id):
        super(Process, self).__init__()
        self.id = id

    def run(self):
        time.sleep(1)
        print("I'm the process with id: {}".format(self.id))
```

要生成进程，必须实例化 `Process` 类并调用方法 `Process.start`。请注意，不直接调用 `Process.run`，而是调用 `Process.start`，它将创建一个新进程，进而调用方法 `Process.run`。要创建并启动新进程，可在上述代码片段末尾添加如下代码行：

```
if __name__ == '__main__':
    p = Process(0)
    p.start()
```

`Process.start` 后面的指令将立即执行，而不是等到进程 `p` 结束后再执行。要等待任务结束，可使用方法 `Process.join`，如下所示。

7

```
if __name__ == '__main__':
    p = Process(0)
    p.start()
    p.join()
```

我们可启动 4 个并行执行的进程。在串行程序中，需要的总时间为 4 秒，但并行执行时，只需要 1 秒。在下面的代码中，我们创建了 4 个并行执行的进程。

```
if __name__ == '__main__':
    processes = Process(1), Process(2), Process(3), Process(4)
    [p.start() for p in processes]
```

请注意，并行进程的执行顺序是无法预测的，它们以什么样的顺序执行取决于操作系统是如何调用的。为验证这一点，你可执行上述程序多次。你将发现每次运行时进程的执行顺序都不同。

模块 multiprocessing 暴露了一个便利的接口，让你能够轻松地给驻留在 multiprocessing.Pool 类中的进程分配任务。

multiprocessing.Pool 类生成一组进程（称为**工作进程**）。要提交任务，可使用这个类的方法 apply/apply_async 和 map/map_async。

方法 Pool.map 对列表中的每个元素执行指定的函数，并返回一个包含结果的列表，其用法与内置（串行）函数 map 相同。

要使用并行映射（map），必须先初始化一个 multiprocessing.Pool 对象。它将工作进程数作为第一个参数；如果没有指定，这个参数将为系统包含的内核数量。要初始化 multiprocessing.Pool 对象，可像下面这样做：

```
pool = multiprocessing.Pool()
pool = multiprocessing.Pool(processes=4)
```

下面来使用 pool.map。如果你有一个计算平方的函数，可将其应用于列表，方法是调用 Pool.map，并将函数和输入列表作为参数传递给它，如下所示。

```
def square(x):
    return x * x

inputs = [0, 1, 2, 3, 4]
outputs = pool.map(square, inputs)
```

函数 Pool.map_async 与 Pool.map 相同，但返回一个 AsyncResult 对象，而不是实际结果。我们调用 Pool.map 时，主程序将停止执行，直到所有工作进程处理完毕。使用 map_async 时，将立即返回一个 AsyncResult 对象，而不阻塞主程序，因此计算是在后台进行的。接下来，我们可随时使用方法 AsyncResult.get 来获取结果，如下所示。

```
outputs_async = pool.map_async(square, inputs)
outputs = outputs_async.get()
```

Pool.apply_async 将由单个函数组成的任务分配给一个工作进程，它将这个函数及其参数作为参数，并返回一个 AsyncResult 对象。可使用 apply_async 来获得类似于使用 map 的效果，如下所示。

```
results_async = [pool.apply_async(square, i) for i in range(100))]
results = [r.get() for r in results_async]
```

### 7.2.2 接口 **Executor**

从 Python 3.2 起，就可使用模块 concurrent.futures 中的接口 Executor 来并行地执行 Python 代码。前一章介绍如何使用 ThreadPoolExecutor 来同时执行多个任务时，你见过接口 Executor。本节将演示 ProcessPoolExecutor 类的用法。

ProcessPoolExecutor 暴露的接口非常简单，至少相比于功能强大的 multiprocessing.Pool 来说如此。实例化 ProcessPoolExecutor 的方式与 ThreadPoolExecutor 类似，只需通过参数 max_workers 传入工作线程数量即可（这个参数默认为可用的 CPU 内核数量）。ProcessPoolExecutor 的主要方法是 submit 和 map。

方法 submit 将一个函数作为参数，并返回一个 Future（参见前一章），用于跟踪提交的函数的执行情况。方法 map 类似于函数 Pool.map，但返回一个迭代器，而不是一个列表。

```
from concurrent.futures import ProcessPoolExecutor

executor = ProcessPoolExecutor(max_workers=4)
fut = executor.submit(square, 2)
# 结果：
# <Future at 0x7f5b5c030940 state=running>

result = executor.map(square, [0, 1, 2, 3, 4])
list(result)
# 结果：
# [0, 1, 4, 9, 16]
```

要从一个或多个 Future 实例中提取结果，可使用函数 concurrent.futures.wait 和 concurrent.futures.as_completed。函数 wait 将一个 future 列表作为参数，并阻塞程序执行，直到所有 future 都执行完毕。然后，就可使用方法 Future.result 来提取结果了。函数 as_completed 也将一个函数作为参数，但返回一个包含结果的迭代器。

```
from concurrent.futures import wait, as_completed

fut1 = executor.submit(square, 2)
fut2 = executor.submit(square, 3)
wait([fut1, fut2])
# 然后就可使用 fut1.result()和 fut2.result()来提取结果了

results = as_completed([fut1, fut2])
```

```
list(results)
# 结果:
# [4, 9]
```

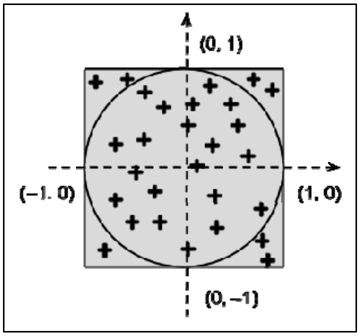
另外，你可使用函数 `asyncio.run_in_executor` 来生成 future，并使用 `asyncio` 库提供的工具和语法来操作结果，这样可同时实现并发和并行。

### 7.2.3　使用蒙特卡洛方法计算 *pi* 的近似值

作为一个示例，我们将实现一个高度并行的程序——使用蒙特卡洛方法计算 *pi* 的近似值。假设有一个边长为 2 单位的正方形，其面积为 4。接下来，我们在这个正方形内雕刻出一个半径为 1 单位的圆。圆的面积为 $pi \times r^2$。将 $r$ 的值代入这个方程，将得到这个圆的面积：$pi \times (1)^2 = pi$。有关上述描述的图形表示，请参阅下图。

如果我们向这个图随机地射击，有些子弹将落在圆内，我们称之为**打中**了，而其他的子弹落在圆外，即**没有打中**。圆的面积与打中的次数成正比，而正方形的面积与射击次数成正比。要计算 *pi* 的值，只需将圆的面积（等于 *pi*）除以正方形的面积（等于 4）即可：

```
hits/total = area_circle/area_square = pi/4
pi = 4 * hits/total
```




在这个程序中，我们将采取如下策略：

- 生成大量均匀分布的随机数$(x, y)$，这些随机数的范围为$(-1, 1)$；
- 检查这些数字是否落在圆内，方法是检查 $x^2 + y^2 \leqslant 1$。

编写并行程序时，首先要做的是编写串行版本，并核实它能够正确地工作。在实际工作中，应将并行化作为优化过程的最后一步。首先，我们需要找出运行速度缓慢的部分；其次，并行化是项耗时的工作，其速度提升受制于处理器数量。这个程序的串行版本的实现如下：

```
import random

samples = 1000000
```

```
hits = 0

for i in range(samples):
    x = random.uniform(-1.0, 1.0)
    y = random.uniform(-1.0, 1.0)

    if x**2 + y**2 <= 1:
        hits += 1

pi = 4.0 * hits/samples
```

计算结果的精度随样本数量的增加而提高。注意，各个循环迭代是彼此独立的——这个问题是高度并行的。

要并行化这些代码，可编写一个函数——`sample`，它对应于单次是否击中的检查。如果样本击中了圆，这个函数将返回 1，否则返回 0。通过运行 `sample` 多次，并将其返回的结果累加，就可得到总共击中了多少次。我们可像下面这样使用 `apply_async` 在多个进程中运行 `sample` 并获取结果。

```
def sample():
    x = random.uniform(-1.0, 1.0)
    y = random.uniform(-1.0, 1.0)

    if x**2 + y**2 <= 1:
        return 1
    else:
        return 0

pool = multiprocessing.Pool()
results_async = [pool.apply_async(sample) for i in range(samples)]
hits = sum(r.get() for r in results_async)
```

可将这两个版本分别放在函数 `pi_serial` 和 `pi_apply_async` 中（这些函数的实现可在文件 pi.py 中找到），并测量它们的执行速度，如下所示。

```
$ time python -c 'import pi; pi.pi_serial()'
real    0m0.734s
user    0m0.731s
sys     0m0.004s
$ time python -c 'import pi; pi.pi_apply_async()'
real    1m36.989s
user    1m55.984s
sys     0m50.386
```

上述基准测试结果表明，第一个并行版本实际上降低了代码的执行速度。这是因为与将任务发送并分配给工作进程的开销相比，执行计算花费的时间很短。

要解决这个问题，必须让开销相比于计算时间可以忽略不计。例如，可让每个工作进程每次处理多个样本，从而降低通信开销。我们可编写一个 `sample_multiple` 函数，它执行多个是否

7

击中的检查，同时修改当前的并行版本，将问题分成 10 个子单元，如下面的代码所示。

```
def sample_multiple(samples_partial):
    return sum(sample() for i in range(samples_partial))

n_tasks = 10
chunk_size = samples/n_tasks
pool = multiprocessing.Pool()
results_async = [pool.apply_async(sample_multiple, chunk_size)
                 for i in range(n_tasks)]
hits = sum(r.get() for r in results_async)
```

我们可将这些代码放在一个名为 `pi_apply_async_chunked` 的函数中，再运行它，如下所示。

```
$ time python -c 'import pi; pi.pi_apply_async_chunked()'
real    0m0.325s
user    0m0.816s
sys     0m0.008s
```

结果好得多，我们将程序的速度提高了一倍多。另外，注意指标 `user` 大于 `real`。为何总 CPU 时间会大于总时间呢？因为有多个 CPU 在同时工作。如果你增大样本数，将发现通信时间与计算时间的比值随之下降，速度得到了进一步的提升。

高度并行的问题处理起来非常简单，但在有些情况下，必须在进程间共享数据。

### 7.2.4　同步和锁

虽然 `multiprocessing` 使用的是进程（这些进程有自己的内存区域），但它也允许你将变量和数组定义为共享内存。要定义共享变量，可使用 `multiprocessing.Value`，并传入一个表示变量数据类型的字符串（`i` 表示整型，`d` 表示 double，`f` 表示 float 等）。要修改这种变量的内容，可使用属性 `value`，如下面的代码所示。

```
shared_variable = multiprocessing.Value('f')
shared_variable.value = 0
```

使用共享内存时，必须考虑同时访问的问题。假设你有一个共享的整型变量，而每个进程都将其值递增多次。你将像下面这样定义一个进程类：

```
class Process(multiprocessing.Process):

    def __init__(self, counter):
        super(Process, self).__init__()
        self.counter = counter

    def run(self):
        for i in range(1000):
            self.counter.value += 1
```

你可在主程序中初始化这个共享变量，并将其传递给 4 个进程，如下面的代码所示。

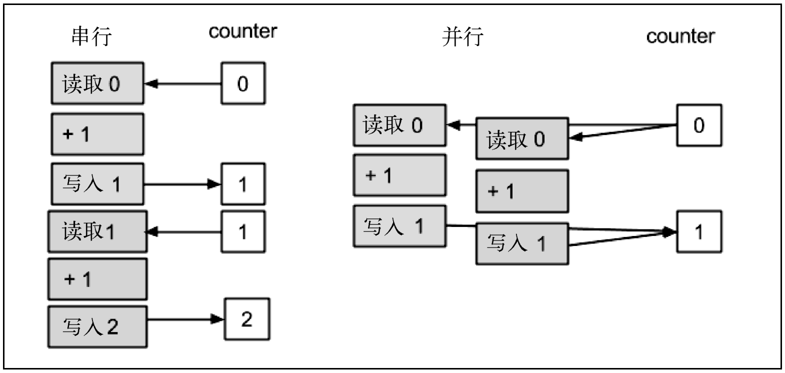
```
def main():
    counter = multiprocessing.Value('i', lock=True)
    counter.value = 0

    processes = [Process(counter) for i in range(4)]
    [p.start() for p in processes]
    [p.join() for p in processes] # 进程执行完毕
    print(counter.value)
```

如果你运行这个程序（目录 code 中的 shared.py），将发现 counter 的最终值不是 4000，而是随机的（在我的机器上，为 2000~2500）。如果我们假定算术运算正确无误，就可确定并行化存在问题。

实际发生的情况是，多个进程同时试图访问同一个共享变量。为搞明白这种情况，请看下图。在串行执行中，第一个进程读取变量的值（数字 0），将其加 1，再将新值（1）写回；第二个变量读取这个新值（1），将其加 1，并将结果（2）写回。

在并行执行中，两个进程同时读取（0），将其加 1，再将结果（1）写回，导致最终的答案不正确。




7

要解决这个问题，需要同步对这个变量的访问，确保每次只有一个进程访问该变量、将其值加 1 并写回。multiprocessing.Lock 类提供了这种功能。要获取和释放锁，可分别使用方法 acquire 和 release，也可将锁用作上下文管理器。由于每次只有一个进程能够获取锁，这种方法可防止多个进程同时执行受保护的代码部分。

我们可定义一个全局锁，并将其用作上下文管理器，以限制对变量 counter 的访问，如下面的代码所示。

```
lock = multiprocessing.Lock()

class Process(multiprocessing.Process):

    def __init__(self, counter):
```

```
        super(Process, self).__init__()
        self.counter = counter

    def run(self):
        for i in range(1000):
            with lock: # 获取锁
                self.counter.value += 1
            # 释放锁
```

诸如锁等同步元语对解决众多问题来说必不可少，但应尽可能少使用，以改善程序的性能。

模块 multiprocessing 还提供了其他通信和同步工具，详情请参阅官方文档（http://docs.python.org/3/library/multiprocessing.html）。

## 7.3　使用 OpenMP 编写并行的 Cython 代码

Cython 通过 OpenMP 提供了一个便利的接口，让你能够实现共享内存式并行处理。这让你能够直接使用 Cython 编写效率极高的并行代码，而无须创建 C 语言包装器。

OpenMP 是一个规范兼 API，设计用于编写多线程并行程序。OpenMP 规范包括一系列 C 语言预处理器指令，用于管理线程以及提供通信模式、负载均衡和其他同步功能。包括 GCC 在内的多个 C/C++和 Fortran 编译器都实现了 OpenMP API。

下面通过一个简单的示例来介绍 Cython 并行功能。Cython 通过模块 cython.parallel 提供了一个基于 OpenMP 的简单 API。要实现并行，最简单的方式是使用 prange，这是一个自动将循环操作分配给多个线程的结构。

首先，我们编写一个程序的串行版本，这个程序计算一个 NumPy 数组中每个元素的平方（参见文件 hello_parallel.pyx）。我们定义了一个函数——square_serial，它将一个缓冲区（buffer）作为输入，并使用这个输入数组中各个元素的平方填充一个输出数组，如下面的代码所示。

```
import numpy as np

def square_serial(double[:] inp):
    cdef int i, size
    cdef double[:] out
    size = inp.shape[0]
    out_np = np.empty(size, 'double')
    out = out_np

    for i in range(size):
        out[i] = inp[i]*inp[i]

    return out_np
```

对于这个遍历数组元素的循环，要实现其并行版本，需要将所有的 rang 调用都替换为

prange。需要注意的是，要使用 prange，必须确保循环体不使用解释器。前面说过，我们需要释放 GIL，而解释器调用通常会获取 GIL，因此要使用线程，必须避免解释器调用。

在 Cython 中，要释放 GIL，可使用上下文 nogil，如下所示。

```
with nogil:
    for i in prange(size):
        out[i] = inp[i]*inp[i]
```

也可使用 prange 选项 nogil=True，这将自动将循环体放在一个 nogil 块中。

```
for i in prange(size, nogil=True):
    out[i] = inp[i]*inp[i]
```

在 prange 块中试图调用 Python 代码将引发错误。禁止的操作包括函数调用、对象初始化等。要允许在 prange 块中执行这些操作（这可能是为了调试），必须使用 gil 语句重新启用 GIL。

```
for i in prange(size, nogil=True):
    out[i] = inp[i]*inp[i]
    with gil:
        x = 0 # Python 赋值
```

现在可以将这些代码作为 Python 扩展模块进行编译，以便测试它们。要启用 OpenMP 支持，必须修改文件 setup.py，在其中包含编译选项 -fopenmp。为此，可使用 distutils 中的 distutils.extension.Extension 类，并将它传递给 cythonize。下面是完整的 setup.py 文件。

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Build import cythonize

hello_parallel = Extension('hello_parallel',
                           ['hello_parallel.pyx'],
                            extra_compile_args=['-fopenmp'],
                            extra_link_args=['-fopenmp'])

setup(
    name='Hello',
    ext_modules = cythonize(['cevolve.pyx', hello_parallel]),
)
```

通过使用 prange，可轻松地并行化 Cython 版的 ParticleSimulator。下面的代码包含第 4 章编写的 Cython 模块 cevolve.pyx 中的函数 c_evolve。

```
def c_evolve(double[:, :] r_i,double[:] ang_speed_i,
             double timestep,int nsteps):

    # cdef 声明

    for i in range(nsteps):
        for j in range(nparticles):
            # 循环体
```

首先，反转循环的顺序，让外面的循环并行地执行（迭代之间彼此独立）。由于粒子之间没有交互，因此修改迭代顺序不会有任何问题，如下面的代码所示。

```
for j in range(nparticles):
    for i in range(nsteps):

        # 循环体
```

接下来，将外部循环中的 range 调用替换为 prange，并将获取 GIL 的调用删除。由于已经使用静态类型改进了代码，因此可安全地使用 nogil 选项，如下所示。

```
for j in prange(nparticles, nogil=True)
```

现在可以将这些函数包装到函数 benchmark 中，以便对它们进行比较并评估性能方面的改进了。

```
In [3]: %timeit benchmark(10000, 'openmp') # Running on 4 processors
1 loops, best of 3: 599 ms per loop
In [4]: %timeit benchmark(10000, 'cython')
1 loops, best of 3: 1.35 s per loop
```

有趣的是，通过使用 prange 编写一个并行版本，获得了两倍的速度提升。

## 7.4　并行自动化

前面说过，常规 Python 程序因 GIL 无法实现线程并行化。到目前为止，我们都是使用独立的进程来避开这种问题，但相比于启动线程，启动进程需要的时间和内存要多得多。

我们还看到，通过避开 Python 环境，我们将原本就很快的 Cython 代码的速度又提高了两倍。这种策略可实现轻量级的并行，但多了一个额外的编译步骤。本节将通过特殊库进一步探索这种策略，这些特殊库能够将代码自动转换为并行版本，从而高效地执行。

当前，实现了并行自动化的包包括你熟悉的 JIT 编译器 numexpr 和 Numba。还有一些包能够自动优化和并行化数组和矩阵密集型表达式，它们对数值计算和机器学习应用程序来说至关重要。

Theano 是一个项目，让你能够定义包含数组的数学表达式（更笼统地说就是**张量**），并将它们编译成快速语言，如 C 或 C++。Theano 实现的很多操作都是可并行化的，并可在 CPU 和 GPU 中运行。

Tensorflow 是一个类似于 Theano 的库，也是为计算数组密集型数学表达式而设计的，但不会将表达式转换为特殊的 C 语言代码，而是在高效的 C++引擎中执行操作。

在要解决的问题可用一串矩阵和基于元素的运算（如**神经网络**）表示时，Theano 和 Tensorflow 都是理想的选择。

### 7.4.1 Theano 初步

Theano 有点像编译器，但还能表示、操作和优化数学表达式，同时能够在 CPU 和 GPU 中运行代码。从 2010 年起，Theano 就一直在不断推出改进版本，并被其他几个 Python 项目用来自动生成高效的计算模型。

在 Theano 中，首先需要**定义**要运行的函数，方法是使用一个纯粹的 Python API 来指定变量和变换。然后，这些定义将被编译成机器码进行执行。

在本节的第一个示例中，我们将探索如何实现一个计算平方的函数。我们用一个标量变量（a）表示输入，再进行变换以获得其平方值（用 a_sq 表示）。在下面的代码中，我们使用函数 T.scalar 定义这个变量，并使用常规运算符**来获得一个新变量。

```
import theano.tensor as T
import theano as th
a = T.scalar('a')
a_sq = a ** 2
print(a_sq)
# 输出：
# Elemwise{pow,no_inplace}.0
```

如你所见，没有计算具体的值，执行的变换是纯粹的符号。要使用这个变换，需要生成一个函数。为此，可使用实用工具 th.function，它接受两个参数，分别是输入变量列表和输出变换（这里是 a_sq）。

```
compute_square = th.function([a], a_sq)
```

Theano 将花时间将这个表达式转换为高效的 C 语言代码，并对代码进行编译，而所有这些操作都是在幕后进行的！th.function 的返回值是一个可直接使用的 Python 函数，下面的代码演示了如何使用这个返回的函数。

```
compute_square(2)
4.0
```

compute_square 正确地返回了输入值的平方，这没什么可奇怪的。然而，注意返回的值并不是整数（与输入类型一样），而是浮点数。这是因为在 Theano 中，变量的类型默认为 float64。要验证这一点，可查看变量 a 的 dtype 属性。

```
a.dtype
# 结果：
# float64
```

相比于 Numba，Theano 的行为有天壤之别。Theano 不会编译通用的 Python 代码，也不做任何类型推断；定义 Theano 函数时，必须准确地指定类型。

Theano 真正的威力在于它对数组表达式的支持。要定义一维向量，可使用函数 T.vector，

7

它返回的变量支持广播操作，就像 NumPy 数组一样。例如，我们可计算两个向量对应元素的平方和，如下所示。

```
a = T.vector('a')
b = T.vector('b')
ab_sq = a**2 + b**2
compute_square = th.function([a, b], ab_sq)

compute_square([0, 1, 2], [3, 4, 5])
# 结果：
# array([ 9., 17., 29.])
```

这里的理念是将 Theano API 作为一种微型语言，用来合并各种 Numpy 数组表达式，这样将生成高效的机器码。

> Theano 的一个卖点是能够简化算法和自动计算梯度，更详细的信息请参阅官方文档（http://deeplearning.net/software/theano/introduction.html）。

为通过一个熟悉的用例来演示 Theano 的功能，我们再次来并行地计算 *pi* 的近似值。这个函数将两组随机坐标作为输入，并返回 *pi* 的近似值。对于输入的随机数，我们将其定义为向量 `x` 和 `y`。为检查它们是否在圆内，我们使用基于元素的标准操作，并将这个表达式存储在变量 `hit_test` 中。

```
x = T.vector('x')
y = T.vector('y')

hit_test = x ** 2 + y ** 2 < 1
```

现在需要计算 `hit_test` 中值为 `True` 的元素个数，为此可计算 `hit_test` 的所有元素的和（将隐式把元素的值转换为整数）。要计算 *pi* 的近似值，需要计算击中次数和射击次数的比值。需要执行的计算如下面的代码所示。

```
hits = hit_test.sum()
total = x.shape[0]
pi_est = 4 * hits/total
```

为测量这种 Theano 实现的执行时间，可使用 `th.function` 和模块 `timeit`。在这里的测试中，我们传入两个长度为 30 000 的数组，并使用 `timeit.timeit` 多次执行函数 `calculate_pi`。

```
calculate_pi = th.function([x, y], pi_est)

x_val = np.random.uniform(-1, 1, 30000)
y_val = np.random.uniform(-1, 1, 30000)

import timeit
res = timeit.timeit("calculate_pi(x_val, y_val)",
"from __main__ import x_val, y_val, calculate_pi", number=100000)
print(res)
# 输出：
# 10.905971487998613
```

串行执行这个函数时，花费了大约 10 秒的时间。Theano 能够自动并行化代码——使用 OpenMP 和 BLAS（Basic Linear Algebra Subprograms）线性代数例程等专用包实现基于元素的操作和矩阵操作。要启用并行执行，可使用配置选项。

在 Theano 中，要设置配置选项，可在导入时修改对象 theano.config 中变量的值。例如，要启用 OpenMP 支持，可执行如下命令：

```
import theano
theano.config.openmp = True
theano.config.openmp_elemwise_minsize = 10
```

与 OpenMP 相关的参数如下。

- openmp_elemwise_minsize：这是一个整数，表示仅当数组长度超过多少时，才对基于元素的操作启用并行化（数组太小时，并行化的开销可能降低性能）。
- openmp：这是一个布尔标志，决定是否激活 OpenMP 编译（默认应激活）。

要控制分配给 OpenMP 执行的线程数，可在执行代码前设置环境变量 OMP_NUM_THREADS。

现在可编写一个简单的基准测试程序，来演示如何使用 OpenMP。我们将 *pi* 值估算示例的代码放在文件 test_theano.py 中。

```
# 文件：test_theano.py
import numpy as np
import theano.tensor as T
import theano as th
th.config.openmp_elemwise_minsize = 1000
th.config.openmp = True

x = T.vector('x')
y = T.vector('y')

hit_test = x ** 2 + y ** 2 <= 1
hits = hit_test.sum()
misses = x.shape[0]
pi_est = 4 * hits/misses

calculate_pi = th.function([x, y], pi_est)
x_val = np.random.uniform(-1, 1, 30000)
y_val = np.random.uniform(-1, 1, 30000)

import timeit
res = timeit.timeit("calculate_pi(x_val, y_val)",
                    "from __main__ import x_val, y_val,
                    calculate_pi", number=100000)
print(res)
```

现在可从命令行运行这些代码，并通过设置环境变量来增加线程数，以评估其可伸缩性。

```
$ OMP_NUM_THREADS=1 python test_theano.py
10.905971487998613
$ OMP_NUM_THREADS=2 python test_theano.py
7.538279129999864
$ OMP_NUM_THREADS=3 python test_theano.py
9.405846934998408
$ OMP_NUM_THREADS=4 python test_theano.py
14.634153957000308
```

有趣的是，使用两个线程时，性能有小幅的提升，但再增加线程数时，性能急剧下降。这意味着就这里的输入规模而言，使用两个以上的线程没有任何好处，因为启动新线程和同步其共享数据的代价比并行计算带来的好处大。

要获得良好的并行性能需要一定的技巧，因为这取决于具体的操作以及它们访问底层数据的方式。一般而言，对并行程序的性能进行测量至关重要，而要获得大幅的速度提升，需要反复试验。

例如，只要稍微修改一下代码，并行性能就会急剧下降。前面检查是否击中时，我们直接使用了方法 `sum`，这依赖于布尔数组 `hit_tests` 的隐式转换。如果我们执行显式转换，Theano 生成的代码将稍微不同，导致多线程带来的好处更小。可修改文件 test_theano.py 来验证这一点：

```
# 旧版本
# hits = hit_test.sum()
hits = hit_test.astype('int32').sum()
```

如果你再次运行基准测试程序，将发现线程的多少对运行时间没有太大的影响，但相比于原来的版本，速度得到了极大的提高。

```
$ OMP_NUM_THREADS=1 python test_theano.py
5.822126664999814
$ OMP_NUM_THREADS=2 python test_theano.py
5.697357518001809
$ OMP_NUM_THREADS=3 python test_theano.py
5.636914656002773
$ OMP_NUM_THREADS=4 python test_theano.py
5.764030176000233
```

### 剖析 Theano 代码

鉴于性能测量和分析的重要性，Theano 提供了功能强大且信息丰富的剖析工具。要生成剖析数据，只需给 `th.function` 添加选项 `profile=True` 即可。

```
calculate_pi = th.function([x, y], pi_est, profile=True)
```

剖析器将在函数运行时收集数据（如通过 `timeit` 或直接调用）。要打印剖析摘要，可执行命令 `summary`，如下所示。

```
calculate_pi.profile.summary()
```

为生成剖析数据，我们在脚本中添加选项 profile=True，并再次运行它（在这里，我们将环境变量 OMP_NUM_THREADS 设置为 1）。另外，我们还将这个脚本恢复到隐式转换 hit_tests 的版本。

> 你也可使用选项 config.profile 全局地设置剖析。

calculate_pi.profile.summary()打印的输出很长，包含大量的信息，下面是其中的一部分。输出由包含时间信息的三部分组成，依次为 Class、Ops 和 Apply。这里将重点放在 Ops 部分；Ops 大致相当于编译后的 Theano 代码中使用的函数。如你所见，大约 80%的时间都花在计算两个数的平方和上，而其他时间花在计算元素的和上。

```
Function profiling
==================
  Message: test_theano.py:15

... other output
Time in 100000 calls to Function.__call__: 1.015549e+01s
... other output

Class
---
<% time> <sum %> <apply time> <time per call> <type> <#call> <#apply>
<Class name>
.... timing info by class

Ops
---
<% time> <sum %> <apply time> <time per call> <type> <#call> <#apply> <Op
name>
  80.0%    80.0%       6.722s       6.72e-05s     C     100000       1
Elemwise{Composite{LT((sqr(i0) + sqr(i1)), i2)}}
  19.4%    99.4%       1.634s       1.63e-05s     C     100000       1
Sum{acc_dtype=int64}
   0.3%    99.8%       0.027s       2.66e-07s     C     100000       1
Elemwise{Composite{((i0 * i1) / i2)}}
   0.2%   100.0%       0.020s       2.03e-07s     C     100000       1
Shape_i{0}
   ... (remaining 0 Ops account for 0.00%(0.00s) of the runtime)

Apply
------
<% time> <sum %> <apply time> <time per call> <#call> <id> <Apply name>
... timing info by apply
```

这与我们在第一个基准测试程序中发现的情况一致。使用两个线程时，代码的执行时间从 11 秒缩短到大约 8 秒。根据这些数字，可分析时间都花在了什么地方。

在这 11 秒中，80%（大约 8.8 秒）花在执行基于元素的操作上。这意味着在完美的并行条件

下，使用两个线程时时间将缩短 4.4 秒，即从理论上说，此时的执行时间将为 6.6 秒。考虑到实际测量到的执行时间大约为 8 秒，看起来使用线程会带来一些额外开销（1.4 秒）。

### 7.4.2 Tensorflow

Tensorflow 也是一个设计用于快速执行数值计算和并行自动化的库，这是 Google 于 2015 年发布的开源项目。Tensorflow 像 Theano 那样创建数学表达式，但不将表达式编译成机器码，而是在使用 C++编写的外部引擎中执行它们。Tensorflow 支持在一个或多个 CPU 和 GPU 中执行和部署并行代码。

Tensorflow 的用法与 Theano 很像。要在 Tensorflow 中创建变量，可使用函数 `tf.placeholder`，它将一个数据类型作为输入。

```
import tensorflow as tf

a = tf.placeholder('float64')
```

在 Tensorflow 中，定义数学表达式的方式与 Theano 很像，但命名约定有些不同，对 NumPy 语义的支持也更有限。

Tensorflow 不像 Theano 那样将函数编译成 C 语言代码，再编译成机器码，而是将定义的数学函数序列化（包含变量和变换的数据结构被称为**计算图**），再在特定的设备上执行它们。要配置设备和上下文，可使用 `tf.Session` 对象。

定义所需的表达式后，需要初始化 `tf.Session`，这种对象可用来执行计算图（使用方法 `Session.run`）。下面的示例演示了如何使用 Tensorflow API 来计算相应元素的平方和。

```
a = tf.placeholder('float64')
b = tf.placeholder('float64')
ab_sq = a**2 + b**2

with tf.Session() as session:
    result = session.run(ab_sq, feed_dict={a: [0, 1, 2],
                                           b: [3, 4, 5]})
    print(result)
# 输出：
# array([ 9., 17., 29.])
```

在 Tensorflow 中，并行化是由其智能执行引擎自动实现的；通常，无须做很大的调整，自动实现的并行化的效果就很好。然而，Tensorflow 最适合处理深度学习负载，这种负载包含复杂函数的定义，即使用大量的矩阵乘法以及计算梯度。

下面使用 Tensorflow 再次实现 *pi* 值估算示例，测量其执行速度和并行性，并与 Theano 实现进行比较。我们需要做的工作如下。

- 定义变量 x 和 y，并使用广播操作检查是否击中。
- 使用函数 tf.reduce_sum 计算数组 hit_tests 的元素的和。
- 使用配置选项 inter_op_parallelism_threads 和 intra_op_parallelism_threads 初始化一个 Session 对象。这些选项指定用于执行不同类型的并行操作的线程数。请注意，创建第一个 Session 对象时，使用的这些选项为整个脚本（包括后面的 Session 实例）设置线程数。

我们编写一个名为 test_tensorflow.py 的脚本，它包含如下代码。请注意，线程数是由传递给这个脚本的第一个参数（sys.argv[1]）指定的。

```
import tensorflow as tf
import numpy as np
import time
import sys

NUM_THREADS = int(sys.argv[1])
samples = 30000

print('Num threads', NUM_THREADS)
x_data = np.random.uniform(-1, 1, samples)
y_data = np.random.uniform(-1, 1, samples)

x = tf.placeholder('float64', name='x')
y = tf.placeholder('float64', name='y')

hit_tests = x ** 2 + y ** 2 <= 1.0
hits = tf.reduce_sum(tf.cast(hit_tests, 'int32'))

with tf.Session
    (config=tf.ConfigProto
    (inter_op_parallelism_threads=NUM_THREADS,
     intra_op_parallelism_threads=NUM_THREADS)) as sess:
start = time.time()
for i in range(10000):
    sess.run(hits, {x: x_data, y: y_data})
print(time.time() - start)
```

7

如果运行这个脚本多次，并在每次都给 NUM_THREADS 指定不同的值，将发现性能与 Theano 实现差别不大，且并行化带来的性能提升很有限。

```
$ python test_tensorflow.py 1
13.059704780578613
$ python test_tensorflow.py 2
11.938535928726196
$ python test_tensorflow.py 3
12.783955574035645
$ python test_tensorflow.py 4
12.158143043518066
```

使用诸如 Tensorflow 和 Theano 软件包的主要优点是，它们支持并行地执行机器学习算法中常用的矩阵操作。这很管用，因为在 GPU 硬件上执行时，这些操作的性能将得到极大的提升，原因是 GPU 能够以极高的吞吐量执行这些操作。

### 7.4.3　在 GPU 中运行代码

本节将演示如何将 GPU 同 Theano 和 Tensorflow 结合起来使用。作为示例，我们将测量一个非常简单的矩阵乘法在 GPU 上的执行时间，并将其与在 CPU 上的执行时间进行比较。

> 要完成本节的示例，你需要有 GPU。就学习而言，可使用 Amazon EC2 服务请求一个支持 GPU 的实例。

下面的代码使用 Theano 执行一个简单的矩阵乘法运算。我们使用函数 `T.matrix` 初始化一个二维数组，然后使用方法 `T.dot` 执行矩阵乘法。

```
    from theano import function, config
    import theano.tensor as T
    import numpy as np
    import time

    N = 5000

    A_data = np.random.rand(N, N).astype('float32')
    B_data = np.random.rand(N, N).astype('float32')

    A = T.matrix('A')
    B = T.matrix('B')

    f = function([A, B], T.dot(A, B))

    start = time.time()
    f(A_data, B_data)

    print("Matrix multiply ({}) took {} seconds".format(N, time.time() -
start))
    print('Device used:', config.device)
```

要让 Theano 在 GPU 上执行这些代码，可设置选项 `config.device=gpu`。出于方便考虑，可在命令行使用环境变量 `THEANO_FLAGS` 设置这个配置值，如下所示。将上述代码保存到文件 test_theano_matmul.py 中，然后就可使用下面的命令来测量执行时间了。

```
$ THEANO_FLAGS=device=gpu python test_theano_gpu.py
Matrix multiply (5000) took 0.4182612895965576 seconds
Device used: gpu
```

要在 CPU 上运行这些代码，可使用配置选项 `device=cpu`。

```
$ THEANO_FLAGS=device=cpu python test_theano.py
Matrix multiply (5000) took 2.9623231887817383 seconds
Device used: cpu
```

如你所见，就这个示例而言，在 GPU 上运行时，速度比在 CPU 上运行时快 7.2 倍！

为进行比较，可对使用 Tensorflow 实现的等效代码进行基准测试。Tensorflow 版本的实现如下面的代码片段所示。相比于 Theano 版本，主要不同如下：

- 使用配置管理器 `tf.device` 来指定目标设备（/cpu:0 或/gpu:0）；
- 使用运算符 `tf.matmul` 来执行矩阵乘法。

```
import tensorflow as tf
import time
import numpy as np
N = 5000

A_data = np.random.rand(N, N)
B_data = np.random.rand(N, N)

# 创建一个图

with tf.device('/gpu:0'):
    A = tf.placeholder('float32')
    B = tf.placeholder('float32')
    C = tf.matmul(A, B)

with tf.Session() as sess:
    start = time.time()
    sess.run(C, {A: A_data, B: B_data})
    print('Matrix multiply ({}) took: {}'.format(N, time.time() -
start))
```

如果使用合适的 `tf.device` 选项运行脚本 test_tensorflow_matmul.py，将得到如下执行时间。

```
# 使用 tf.device('/gpu:0')运行
Matrix multiply (5000) took: 1.417285680770874

# 使用 tf.device('/cpu:0')运行
Matrix multiply (5000) took: 2.9646761417388916
```

如你所见，就这个简单示例而言，在 GPU 上运行时性能得到了极大的提升（但没有 Theano 版本那么大）。

要使用 GPU 来自动执行计算，另一种方式是使用你现在应该很熟悉的 Numba。使用 Numba 可将 Python 代码编译成可在 GPU 上运行的程序。这种灵活性让你能够使用简单接口完成高级 GPU 编程，具体地说，Numba 能够让你非常轻松地编写支持 GPU 的泛型通用函数。

在下面的示例中，我们将演示如何编写一个通用函数，它对两个数字执行指数函数并将结果相加。第 5 章说过，这可使用函数 `nb.vectorize` 来实现（我们还显式地将目标设备指定为 CPU）。

```
import numba as nb
import math
@nb.vectorize(target='cpu')
def expon_cpu(x, y):
    return math.exp(x) + math.exp(y)
```

要编译通用函数 `expon_cpu`，以便在 GPU 设备上运行，可使用选项 `target='cuda'`。另外，对于 CUDA 通用函数，还必须输入类型。`expon_gpu` 的实现如下：

```
@nb.vectorize(['float32(float32, float32)'], target='cuda')
def expon_gpu(x, y):
    return math.exp(x) + math.exp(y)
```

现在可以对这两个函数进行基准测试了：将它们应用于两个长度为 1 000 000 的数组。另外请注意，测量执行时间前，我们先调用一次函数，以触发 Numba 即时编译。

```
import numpy as np
import time

N = 1000000
niter = 100

a = np.random.rand(N).astype('float32')
b = np.random.rand(N).astype('float32')

# 触发编译
expon_cpu(a, b)
expon_gpu(a, b)

# 测量时间
start = time.time()
for i in range(niter):
    expon_cpu(a, b)
print("CPU:", time.time() - start)

start = time.time()
for i in range(niter):
    expon_gpu(a, b)
print("GPU:", time.time() - start)
# 输出：
# CPU: 2.4762887954711914
# GPU: 0.8668839931488037
```

在 GPU 上执行时，速度比在 CPU 上执行时提高了 3 倍。请注意，将数据传输给 GPU 的开销非常高，因此仅当数组非常大时，在 GPU 上执行才有优势。

## 7.5　小结

对大型数据集来说，并行处理是一种改善性能的有效方式。高度并行的问题非常适合采用并行处理；对于这种问题，实现并行处理很容易，同时性能可得到极大的提升。

本章介绍了 Python 并行编程的基础知识。你学习了如何使用 Python 标准库中的工具来生成进程，以避开 Python 线程技术的局限性，还学习了如何使用 Cython 和 OpenMP 来实现多线程程序。

对于更复杂的问题，你学习了如何使用 Theano、Tensorflow 和 Numba 包来自动编译数组密集型表达式，以便在 CPU 和 GPU 设备上并行地执行。

下一章将介绍如何使用 Dask 和 PySpark 等库编写在多个处理器和计算机上执行的并行程序。

第 8 章

# 分布式处理

前一章介绍了并行处理的概念以及如何利用多核处理器和 GPU，现在我们再进一步，将注意力转向分布式处理——通过在多台计算机中执行任务来解决问题。

本章将阐述在计算机集群中运行代码的挑战、用例和示例。Python 提供了易于使用且可靠的分布式处理包，让我们能够轻松地实现可伸缩的容错代码。

本章介绍如下主题：

- 分布式计算和 MapReduce 模型；
- Dask 有向无环图；
- 使用 Dask 数组、`Bag` 和 `DataFrame` 编写并行代码；
- 使用 Dask distributed 实现分布式并行算法；
- PySpark 简介；
- Spark 弹性分布式数据集和 `DataFrame`；
- 使用 `mpi4py` 执行科学计算。

## 8.1 分布式计算简介

如今，计算机、智能手机等设备在人们的生活中已不可或缺。每天都有海量的数据生成。数十亿人访问互联网上的服务，而公司不间断地收集数据，以便了解用户，进而提供更有针对性的产品和更佳的用户体验。

为处理越来越多的数据，我们面临着严峻的挑战。大型公司和组织常常打造计算机集群，以便存储、处理和分析复杂的大型数据集。在环境科学和医疗保健等数据密集型领域，也会生成类似的数据集。最近，这些大型数据集被称为**大数据**。大数据分析方法通常涉及机器学习、信息检索和可视化。

计算集群在科学计算领域已使用几十年，这些领域的复杂问题研究必须使用在高性能分布式系统中运行的并行算法。为支持这样的应用程序，高校和其他组织提供并管理着用于研究和工程

方面的超级计算机。运行在超级计算机上的应用程序通常专注于数值计算密集型工作负载，如蛋白质和分子模拟、量子力学计算、气候模型等。

只要想一想将数据和计算任务分布到计算机局域网后通信开销的增加情况，分布式系统编程面临的挑战就显而易见。相比于处理器的速度，网络传输的速度慢如蜗牛，因此使用分布式处理时，尽可能减少网络通信显得更加重要。为此，可采用多种不同的策略，它们优先考虑本地数据处理，不到万不得已不传输数据。

分布式处理面临的另一个挑战是，计算机网络通常是不可靠的。考虑到计算集群可能包含数千台计算机，从概率上说，显然经常会有节点出现故障。有鉴于此，分布式系统必须能够妥善地处理节点故障，避免中断当前执行的工作。所幸各公司投入了大量资源来开发容错分布式引擎，它们能够自动处理前述方方面面。

## MapReduce 简介

MapReduce 是一个编程模型，让你能够以特定的方式表示算法，使其能够在分布式系统中高效地执行。MapReduce 模型最初是由 Google 于 2004 年推出的，旨在将数据集分配给不同的计算机，并将本地处理和**集群节点**之间的通信自动化。

那时，MapReduce 框架和一种分布式文件系统——**Google 文件系统**（GFS 或 GoogleFS）协同工作，这种文件系统是为在计算集群中进行数据切片（partition）和复制而设计的。为存储和处理单个节点容纳不下的数据集，切片很有用，而复制确保系统能够妥善地处理故障。当时，Google 结合使用 MapReduce 和 GFS 是为了建立网页索引，但后来 Doug Cutting（当时是 Yahoo! 一名员工）实现了 MapReduce 和 GFS 概念，推出了最初的 **Hadoop 分布式文件系统**（HDFS）和 Hadoop MapReduce。

MapReduce 暴露的编程模型实际上非常简单，其理念是将计算表示为两个非常通用的步骤：**映射**（Map）和**归并**（Reduce）。有些读者可能熟悉 Python 函数 `map` 和 `reduce`，但在 MapReduce 中，Map 和 Reduce 步骤能够表示的操作更多。

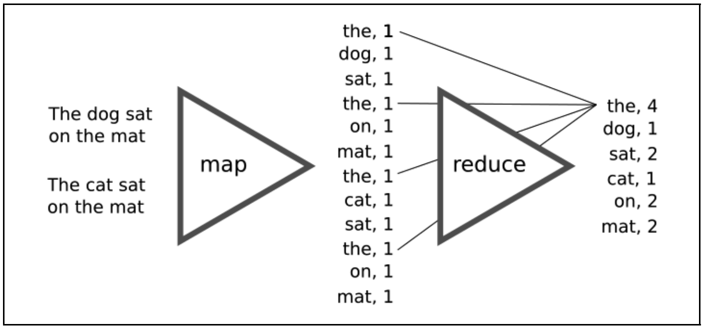
Map 将一组数据作为输入，并对其进行**变换**。通常，Map 的结果是一系列可交给 Reduce 步骤的键–值对；Reduce 步骤聚合键相同的数据项，并对得到的集合应用一个函数，这通常会生成更小的数据集合。

对于前一章介绍的 *pi* 值估算问题，可轻松地将其转换为一系列 Map 和 Reduce 步骤。在这个例子中，输入是一系列随机数字对。变换（Map 步骤）是击中检查，而 Reduce 步骤是计算击中检查结果为 True 的次数。

一个典型的 MapReduce 模型示例是单词计数实现：程序将一系列文档作为输入，并返回每个单词在这些文档中出现的总次数。下图说明了单词计数程序的 Map 和 Reduce 步骤，其中左边是输入文档。Map 操作生成一系列(key, value)项，其中第一个元素为单词，而第二个元素为 1（因

为单词每次出现都将导致最终计数加 1）。

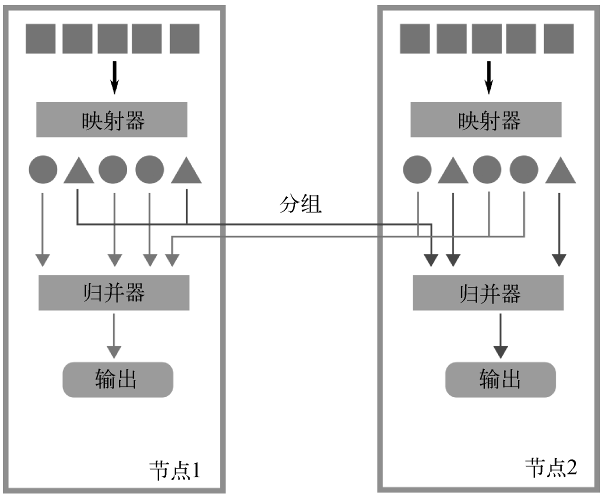
接下来，我们执行 Reduce 操作，将键相同的项聚合起来，得到每个单词出现的总次数。从下图可知，将键为 the 的项的值累积，得到(the, 4)项。




如果我们使用 Map 和 Reduce 操作来实现这种算法，该框架实现将通过巧妙的算法限制节点之间的通信，确保高效地完成数据生成和聚合。

然而，MapReduce 是如何最大限度地减少通信的呢？我们来看一个 MapReduce 任务的完成过程。假设有一个包含两个节点的集群，每个节点都从磁盘加载一个数据分片（通常位于节点本地），为处理数据做好准备。在每个节点中，都创建一个映射器（mapper）进程，并对数据进行处理以生成中间结果。

接下来，必须将数据发送给归并器（reducer）做进一步的处理，但这样做时，必须确保键相同的所有项都被发送给同一个归并器。这项操作被称为**分组**（shuffling），是 MapReduce 模型中最主要的通信任务。



请注意，交换数据前，必须给每个归并器分配一个键子集，这个步骤被称为**切片**（partitioning）。归并器获得自己的键切片后，就可处理数据并将结果写入磁盘了。

通过 Apache Hadoop 项目，MapReduce 框架得以被众多公司和组织广泛使用。最近，推出了一些新框架，它们扩展了 MapReduce 引入的理念，可用于创建这样的系统：能够表示更复杂的工作流程，能够更高效地使用内存，支持精益而高效地执行分布式任务。

在接下来的几节中，我们将介绍在 Python 分布式领域用得最多的两个库：Dask 和 PySpark。

## 8.2 Dask

Dask 是 Continuum Analytics 推出的一个项目（这家公司还推出了 Numba 和包管理器 `conda`），这是一个用于并行和分布式计算的 Python 库，擅长执行数据分析任务，并紧密地集成到了 Python 生态系统中。

Dask 最初用于在单机上处理超过内存量的数据集，但最近随着 Dask Distributed 项目的推出，其代码做了修改，能够在集群中执行任务，且性能和容错功能都极为出色。Dask 支持 MapReduce 型任务以及复杂的数值算法。

### 8.2.1 有向无环图

Dask 背后的理念与前一章介绍的 Theano 和 Tensorflow 的基本思想很像。你可使用一个熟悉的 Python 式 API 来建立执行计划，而这个框架会自动将工作流程划分成任务，并将它们交给多个进程或多台计算机去执行。

Dask 使用**有向无环图**（DAG）来表示变量和操作，而这种图可使用简单的 Python 字典来表示。为了大致演示其中的工作原理，我们将使用 Dask 来计算两个数的和。为定义计算图，我们将输入变量的值存储在字典中，如下所示（这里将变量 `a` 和 `b` 的值都设置为 2）。

```
dsk = {
  "a" : 2,
  "b" : 2,
}
```

每个变量都相当于 DAG 中的一个节点。为创建 DAG，接下来必须定义要对节点执行的操作。在 Dask 中，要定义任务，可在字典（这里为 `dsk`）中添加一个元组，其中包含一个 Python 函数及其位置参数。为实现求和运算，可添加一个名为 `result` 的新节点（你可随便给这个节点命名）。这个节点的值是一个元组，其中包含我们要执行的函数及其参数，如下面的代码所示。 8

```
dsk = {
  "a" : 2,
  "b" : 2,
```

```
  "result": (lambda x, y: x + y, "a", "b")
}
```

出于风格和清晰方面的考虑，可修改求和运算，将 lambda 语句替换为标准库函数 operator.add。

```
from operator import add
dsk = {
  "a" : 2,
  "b" : 2,
  "result": (add, "a", "b")
}
```

必须指出的是，这里使用字符串"a"和"b"指定了要传递给函数的参数，它们表示图中的节点 a 和 b。请注意，前面定义 DAG 时，没有使用任何 Dask 特有的函数，这表明这个框架非常灵活而简洁，因为所有操作都是在简单而熟悉的 Python 字典上执行的。

任务是由调度器执行的。调度器是一个函数，它接受一个 DAG 以及要执行的任务，并返回计算得到的值。默认 Dask 调度器为函数 dask.get，你可像下面这样使用它：

```
import dask

res = dask.get(dsk, "result")
print(res)
# 输出：
# 4
```

所有复杂性都隐藏在调度器背后，调度器会负责将任务分配给不同的线程、进程乃至不同的计算机。调度器 dask.get 采用的是同步串行实现，非常适合用于测试和调试。

就理解 Dask 如何发挥其魔力以及进行调试而言，使用简单的字典来定义 DAG 很有帮助。你还可使用原始（raw）字典来实现 Dask API 中没有的复杂算法。接下来，我们来学习 Dask 是如何通过类似于 NumPy 和 Pandas 的接口来自动生成任务的。

### 8.2.2　Dask 数组

Dask 的主要用途之一是自动生成并行数组操作，这可极大地简化规模超过内存容量的数组的处理工作。Dask 采用的策略是，将数组分割为大量的子单元——Dask 称之为**块**（chunk）。

Dask 在模块 dask.array（以下简称为 da）中实现了一个类似于 NumPy 的数组接口。要从 NumPy 数组创建一个 Dask 数组，可使用函数 da.from_array。这个函数要求你指定块大小，并返回一个 da.array 对象，而 da.array 对象会负责将原始数组分割为指定大小的子单元。在下面的代码中，我们创建了一个包含 30 个元素的数组，并将其分割成块，其中每块包含 10 个元素。

```
import numpy as np
import dask.array as da6

a = np.random.rand(30)

a_da = da.from_array(a, chunks=10)
# 结果:
# dask.array<array-4..., shape=(30,), dtype=float64, chunksize=(10,)>
```

变量 a_da 维护着一个 Dask 图，这个图可通过属性 dask 来访问。为了弄明白 Dask 在幕后做了哪些工作，可查看这个 Dask 图的内容。从下面的示例可知，这个 Dask 图包含 4 个节点，其中一个是源数组，其键为'array-original-4c76'。字典 a_da.dask 中的其他三个键是任务，你可使用它们和函数 dask.array.core.getarray 来访问原始数组中的块。如你所见，每个任务都提取一个包含 10 个元素的切片（slice）。

```
dict(a_da.dask)
# 结果
{('array-4c76', 0): (<function dask.array.core.getarray>,
                     'array-original-4c76',
                     (slice(0, 10, None),)),
 ('array-4c76', 2): (<function dask.array.core.getarray>,
                     'array-original-4c76',
                     (slice(20, 30, None),)),
 ('array-4c76', 1): (<function dask.array.core.getarray>,
                     'array-original-4c76',
                     (slice(10, 20, None),)),
 'array-original-4c76': array([ ... ])
}
```

如果我们对数组 a_da 执行操作，Dask 将生成更多操作块的子任务，这打开了并行的大门。da.array 暴露的接口遵循 NumPy 语义和广播规则。下面的完整代码演示了 Dask 与 NumPy 广播规则、基于元素的操作和其他方法的良好兼容性。

```
N = 10000
chunksize = 1000

x_data = np.random.uniform(-1, 1, N)
y_data = np.random.uniform(-1, 1, N)

x = da.from_array(x_data, chunks=chunksize)
y = da.from_array(y_data, chunks=chunksize)

hit_test = x ** 2 + y ** 2 < 1
hits = hit_test.sum()
pi = 4 * hits / N
```

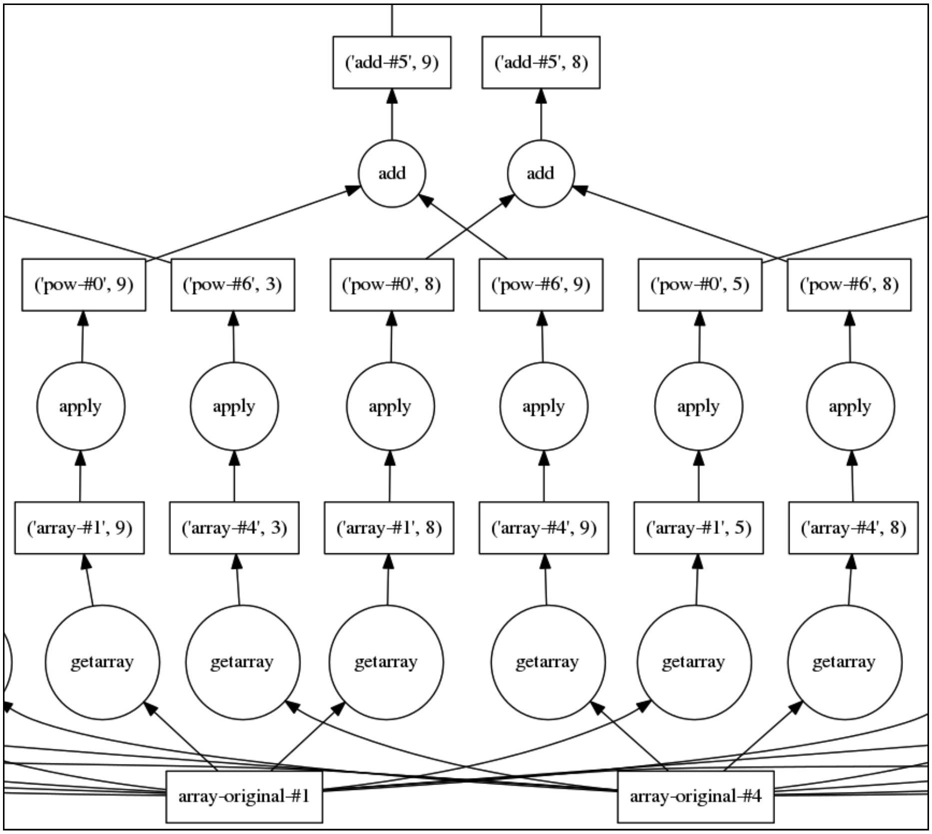


要计算 *pi* 的值，可使用方法 compute。调用这个方法时，也可使用可选参数 get 指定其他调度器（默认情况下，da.array 使用一个多线程调度器）。

```
pi.compute() # 也可这样做：pi.compute(get=dask.get)
```

```
# 结果：
# 3.1804000000000001
```

即便是看起来非常简单的算法，如估算 *pi* 的值，也可能需要执行大量的任务。Dask 提供了计算图可视化工具。下面是 *pi* 值估算的 Dask 图，要获取它，可执行方法 `pi.visualize()`。在这个图中，圆圈表示对节点执行的变换，而节点用矩形表示。通过这个示例，我们可对 Dask 图的复杂性有大致的认识，并知道调度器的职责是制订高效的执行计划，其中包括按正确的顺序排列任务以及挑选出要并行执行的任务。




### 8.2.3　Dask `Bag` 和 `DataFrame`

Dask 提供了其他用于自动生成计算图的数据结构。本节将介绍 `dask.bag.Bag` 和 `dask.dataframe.DataFrame`，其中前者是一种通用的元素集合，可用来编写 MapReduce 式算法代码，而后者是 `pandas.DataFrame` 的分布式版本。

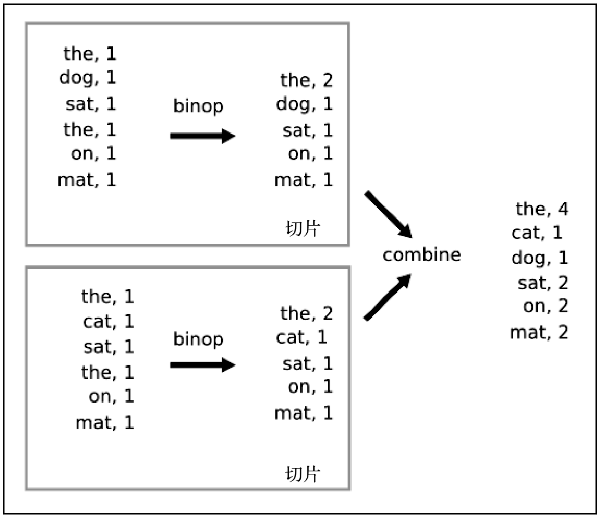
可从 Python 集合轻松地创建 `Bag`。例如，要从列表创建 `Bag`，可使用工厂函数 `from_sequence`。

要指定并行等级，可使用参数 npartitions（这将把 Bag 的内容分成很多块）。在下面的示例中，我们创建了一个 Bag，它包含数字 0~99，并被分成 4 块。

```
import dask.bag as dab
dab.from_sequence(range(100), npartitions=4)
# 结果：
# dask.bag<from_se..., npartitions=4>
```

在下一个示例中，我们将使用类似于 MapReduce 的算法，计算一组字符串中各个单词出现的次数。给定一个序列集合，我们依次使用 str.split 和 concat 来生成一个线性列表，其中包含给定文档中所有的单词。接下来，对于每个单词，我们生成一个字典，其中包含一个单词和值 1（参见本章前面的“MapReduce 简介”一节）。然后，我们编写一个 Reduce 步骤，使用运算符 foldby 来计算单词出现的次数。

变换 foldby 很有用，可用来实现合并单词计数的 Reduce 步骤，这样便无须将元素分组再进行分配。假设我们的单词数据集被分成两个切片。为计算单词出现的次数，一种不错的策略是先计算每个切片中单词出现的次数，再将这些数据合并，得到最终的结果，如下图所示。左边是输入切片。我们先计算每个切片中单词出现的次数（这是使用二元运算 binop 完成的），再使用函数 combine 将这两部分数据合并。




下面的代码演示了如何使用 Bag 和运算符 foldby 来计算单词出现的次数。运算符 foldby 接受 5 个参数。

- key：这是一个函数，返回用于归并操作的键。
- binop：这是一个函数，它接受两个参数——total 和 x。给定总值（到目前为止的累积值），binop 会将下一项合并到总值中。

- initial：这是给 binop 提供的初始累积值。
- combine：这是一个函数，将各个切片的总值合并（这里是简单的求和）。
- initial_combine：这是给 combine 提供的初始累积值。

下面来看看代码：

```
    collection = dab.from_sequence(["the cat sat on the mat",
                                    "the dog sat on the mat"],
npartitions=2)

    binop = lambda total, x: total + x["count"]
    combine = lambda a, b: a + b
    (collection
     .map(str.split)
     .concat()
     .map(lambda x: {"word": x, "count": 1})
     .foldby(lambda x: x["word"], binop, 0, combine, 0)
     .compute())
    # 输出：
    # [('dog', 1), ('cat', 1), ('sat', 2), ('on', 2), ('mat', 2), ('the',
4)]
```

如你所见，要使用 Bag 高效地表示复杂的操作，可能很烦琐。有鉴于此，Dask 提供了另一种数据结构——dask.dataframe.DataFrame，这种数据结构是为分析型工作负载而设计的。在 Dask 中，要初始化 DataFrame，可使用很多方式，如从分布式文件系统中的 CSV 文件初始化，或者直接从 Bag 初始化。就像 da.array 提供了准确反映 NumPy 功能的 API 一样，Dask DataFrame 可作为分布式 pandas.DataFrame 使用。

为了演示这一点，我们将使用 DataFrame 来计算单词出现的次数。我们首先加载数据，以生成一个由单词组成的 Bag，再使用方法 to_dataframe 将这个 Bag 转换为 DataFrame。通过向方法 to_dataframe 传递一个列名，可初始化一个 DataFrame，它只包含一列，名为 words。

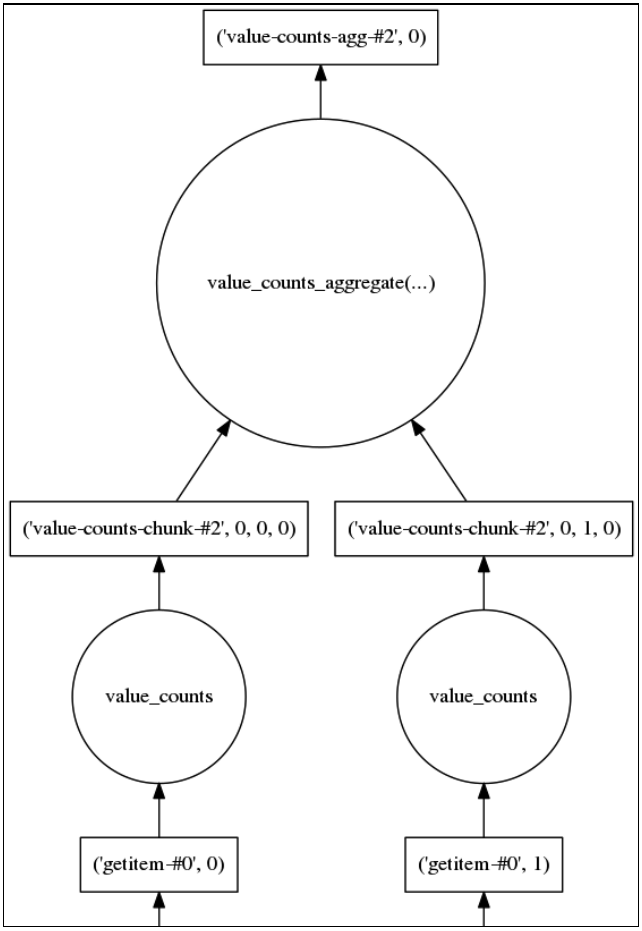
```
    collection = dab.from_sequence(["the cat sat on the mat",
                                    "the dog sat on the mat"],
npartitions=2)
    words = collection.map(str.split).concat()
    df = words.to_dataframe(['words'])
    df.head()
    # 结果：
    #   words
    # 0   the
    # 1   cat
    # 2   sat
    # 3    on
    # 4   the
```

Dask DataFrame 精确地复制了 pandas.DataFrame API。要计算单词出现的次数，只需对 words 列调用方法 value_counts，而 Dask 将自动设计一种并行计算策略。要触发这种计算，

只需调用方法 compute：

```
df.words.value_counts().compute()
# 结果:
# the    4
# sat    2
# on     2
# mat    2
# dog    1
# cat    1
# Name: words, dtype: int64
```

你可能会提出一个有趣的问题：DataFrame 在幕后使用的是什么样的算法？要找出这个问题的答案，可查看生成的 Dask 图的上半部分，如下图所示。最下面的两个矩形表示两个数据集切片，它们被存储为两个 pd.Series 实例。为计算单词出现的总次数，Dask 先对每个 pd.Series 执行 value_counts，再使用 value_counts_aggregate 将次数合并。



如你所见，Dask 数组和 DataFrame 利用了 NumPy 和 Pandas 的快速向量化实现，来获得出色的性能和稳定性。

### 8.2.4 Dask distributed

Dask 项目的最初几个版本被设计成在单机上运行，使用的是基于线程或进程的调度器。最近推出了新的分布式后端实现，可用来在计算机网络上创建和运行 Dask 图。

Dask distributed 不会随 Dask 自动安装，要安装这个库，可使用包管理器 conda（使用命令$ conda install distributed），也可使用 pip（使用命令$ pip install distributed）。

Dask distributed 使用起来非常容易，要完成准备工作，最简单的方式是实例化一个 Client 对象。

```
from dask.distributed import Client

client = Client()
# 结果:
# <Client: scheduler='tcp://127.0.0.1:46472' processes=4 cores=4>
```

默认情况下，Dask 将在本地计算机上启动几个重要的进程。要通过 Client 实例调度和执行分布式任务，这些进程必不可少。Dask 集群的主要组件是一个**调度器**和一系列**工作进程**。

**调度器**是负责将工作分配给工作进程并监视和管理结果的进程。一般而言，任务被提交给用户后，调度器将找到一个空闲的工作进程，并将任务提交给它去执行。工作进程完成任务后，将告诉调度器结果可用了。

工作进程接受到来的任务并生成结果。工作进程可能位于网络中其他的计算机上。工作进程使用 ThreadPoolExecutor 来执行任务；在使用的函数（如 nogil 块中的 Numpy、Pandas 和 Cython 函数）不会获取 GIL 时，这样可实现并行性。执行纯粹的 Python 代码时，启动大量单线程工作进程更有利，因为这样即便代码会获取 GIL，也将实现并行性。

可使用 Client 类中熟悉的异步方法，手动将任务提交给调度器。例如，要将函数提交给集群去执行，可使用方法 Client.map 和 Client.submit。下面的代码演示了如何使用 Client.map 和 Client.submit 来计算几个数字的平方。Client 将向调度器提交一系列任务，对于每个任务，我们都将获得一个 Future 实例。

```
def square(x):
    return x ** 2

fut = client.submit(square, 2)
# 结果:
# <Future: status: pending, key:
```

```
square-05236e00d545104559e0cd20f94cd8ab>

    client.map(square)
    futs = client.map(square, [0, 1, 2, 3, 4])
    # 结果：
    # [<Future: status: pending, key:
square d043f00c1427622a694f518348870a2f>,
    #  <Future: status: pending, key:
square-9352eac1fb1f6659e8442ca4838b6f8d>,
    #  <Future: status: finished, type: int, key:
    #  square-05236e00d545104559e0cd20f94cd8ab>,
    #  <Future: status: pending, key:
    #  square-c89f4c21ae6004ce0fe5206f1a8d619d>,
    #  <Future: status: pending, key:
    #  square-a66f1c13e2a46762b092a4f2922e9db9>]
```

到目前为止，与本书前面使用 `TheadPoolExecutor` 和 `ProcessPoolExecutor` 的情况很像。然而，Dask distributed 不仅提交任务，还将计算结果缓存到工作进程的内存中。在上面的示例中就可看到缓存在发挥作用。我们首次调用 `client.submit` 时，创建了任务 `square(2)`，其状态被设置为**未完**（pending）；我们接着调用 `client.map` 时，任务 `square(2)` 被再次提交给调度器，但这次没有重新计算它的值，调度器直接从工作进程那里获取了结果。因此，map 返回的第三个 `Future` 的状态为完成（finished）。

要从一系列 `Future` 实例中获取结果，可使用方法 `Client.gather`：

```
client.gather(futs)
# 结果：
# [0, 1, 4, 9, 16]
```

`Client` 还可用来运行任何 Dask 图。例如，要估算 *pi* 值，只需将函数 `client.get` 作为可选参数传递给 `pi.compute`。

```
pi.compute(get=client.get)
```

这种特征让 Dask 的可伸缩性极强，因为你可使用较简单的调度器在本地计算机上开发并运行算法，如果对性能不满意，可在由数百台计算机组成的集群上运行这些算法。

### 手动建立集群

要手动实例化调度器和工作进程，可使用命令行工具 `dask-scheduler` 和 `dask-worker`。首先，使用命令 `dask-scheduler` 初始化一个调度器。

```
$ dask-scheduler
distributed.scheduler - INFO - ---------------------------------------------
---
distributed.scheduler - INFO - Scheduler at: tcp://192.168.0.102:8786
distributed.scheduler - INFO - bokeh at: 0.0.0.0:8788
distributed.scheduler - INFO - http at: 0.0.0.0:9786
distributed.bokeh.application - INFO - Web UI:
```

```
http://127.0.0.1:8787/status/
distributed.scheduler - INFO - ----------------------------------------------
---
```

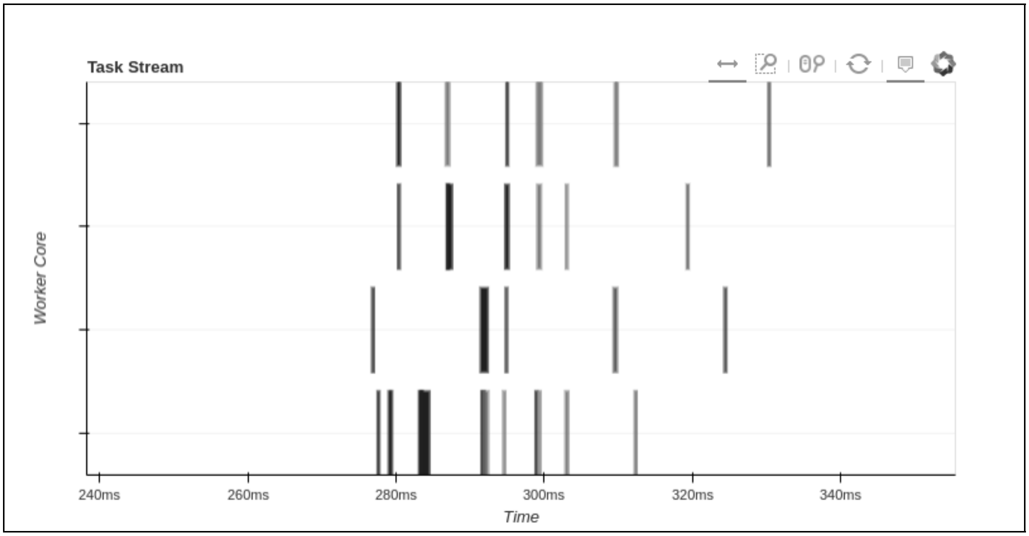
这将给调度器提供一个地址，还将提供一个 Web UI 地址，可通过访问它来监视集群的状态。现在可以给调度器分配一些工作进程了，为此可使用命令 `dask-worker`，并将调度器的地址传递给工作进程。这将自动启动一个包含 4 个线程的工作进程。

```
$ dask-worker 192.168.0.102:8786
distributed.nanny - INFO - Start Nanny at: 'tcp://192.168.0.102:45711'
distributed.worker - INFO - Start worker at: tcp://192.168.0.102:45928
distributed.worker - INFO - bokeh at: 192.168.0.102:8789
distributed.worker - INFO - http at: 192.168.0.102:46154
distributed.worker - INFO - nanny at: 192.168.0.102:45711
distributed.worker - INFO - Waiting to connect to: tcp://192.168.0.102:8786
distributed.worker - INFO - ----------------------------------------------
--
distributed.worker - INFO - Threads: 4
distributed.worker - INFO - Memory: 4.97 GB
distributed.worker - INFO - Local Directory: /tmp/nanny-jh1esoo7
distributed.worker - INFO - ----------------------------------------------
--
distributed.worker - INFO - Registered to: tcp://192.168.0.102:8786
distributed.worker - INFO - ----------------------------------------------
--
distributed.nanny - INFO - Nanny 'tcp://192.168.0.102:45711' starts worker
process 'tcp://192.168.0.102:45928'
```

Dask 调度器的适应能力极强，如果我们先添加再删除一个工作进程，调度器将能够跟踪哪些结果不可用，并根据需要重新计算。最后，要在 Python 会话中使用你初始化的调度器，只需初始化一个 `Client` 实例，并提供调度器的地址。

```
client = Client(address='192.168.0.102:8786')
# 结果：
# <Client: scheduler='tcp://192.168.0.102:8786' processes=1 cores=4>
```

Dask 还提供了便利的用于诊断的 Web UI，可用来监视状态以及在集群上执行的每项任务花费的时间。在下图中，Task Stream 指出了执行 *pi* 值估算花费的时间。图中的每条灰色线对应于工作进程使用的一个线程（在这里，有一个工作进程——也叫 Worker Core，它包含 4 个线程），而每个矩形框对应于一个任务，这些矩形框是彩色的——相同的颜色表示相同类型的任务，如加法运算、求幂或指数运算。从该图可知，所有矩形框都很小且彼此相隔很远，这意味着相比于通信开销，这些任务都很小。




就这个示例而言，提高块大小是有利的，因为这样每项任务的运行时间将相对于通信时间更长。

## 8.3 使用 PySpark

当前，Apache Spark 是最受欢迎的分布式计算项目之一。发布于 2014 年的 Spark 是使用 Scala 编写的，它集成了 HDFS，相比于 Hadoop MapReduce 框架有多个方面的优势和改进。

不同于 Hadoop MapReduce，Spark 设计用于交互地处理数据，并提供了供 Java、Scala 和 Python 编程语言使用的 API。由于 Spark 采用的架构不同，尤其是将结果存储在内存中，其速度通常比 Hadoop MapReduce 快得多。

### 8.3.1 搭建 Spark 和 PySpark 环境

要从头搭建 PySpark 环境，需要安装 Java 和 Scala 运行时，从源代码编译这个项目，并配置 Python 和 Jupyter notebook 以便安装 Spark 时能够使用它们。一种搭建 PySpark 环境的方式是，使用通过 Docker 容器提供的配置好的 Spark 集群，这种方式简单且不容易出错。

> Docker 可从 https://www.docker.com/下载。如果你不熟悉容器，可阅读下一章中有关这方面的简介。

要搭建 Spark 集群，只需切换到本章代码文件所在的目录（其中有一个名为 Dockerfile 的文件），并执行如下命令：

```
$ docker build -t pyspark
```

这个命令将自动下载 Spark、Python 和 Jupyter notebook，并在一个隔离环境中安装和配置它们。要启动 Spark 和 Jupyter notebook 会话，可执行如下命令：

```
$ docker run -d -p 8888:8888 -p 4040:4040 pyspark
22b9dbc2767c260e525dcbc562b84a399a7f338fe1c06418cbe6b351c998e239
```

这个命令打印一个独一无二的 ID（**容器 id**，可用来引用应用程序容器），并在后台启动 Spark 和 Jupyter notebook。选项`-p` 确保我们能够在本地计算机中访问 SparkUI 和 Jupyter 网络端口。执行这个命令后，就可在浏览器中输入地址 http://127.0.0.1:8888 来访问 Jupyter notebook 会话。要检查是否正确地初始化了 Spark，可创建一个新的 notebook，并在其中一个单元格中执行如下代码：

```
import pyspark
sc = pyspark.SparkContext('local[*]')

rdd = sc.parallelize(range(1000))
rdd.first()
# 结果：
# 0
```

这将初始化一个 `SparkContext`，并获取一个集合中的第一个元素（这些新术语将在后面详细解释）。初始化 `SparkContext` 后，还可访问 http://127.0.0.1:4040 来打开 Spark Web UI。

完成搭建工作后，接下来探索 Spark 的工作原理，以及如何使用其功能强大的 API 来实现简单的并行算法。

### 8.3.2　Spark 架构

Spark 集群是一组分布在不同计算机上的进程。**驱动器程序**（driver program）是一个进程，如 Scala 或 Python 解释器，用户使用它来提交要执行的任务。

与 Dask 中一样，用户可使用一个特殊的 API 来创建任务图，并将这些任务提交给**集群管理器**（cluster manager）。集群管理器负责将这些任务分配给**执行器**（executor）——负责执行任务的进程。在多用户系统中，集群管理器还负责给每位用户分配资源。

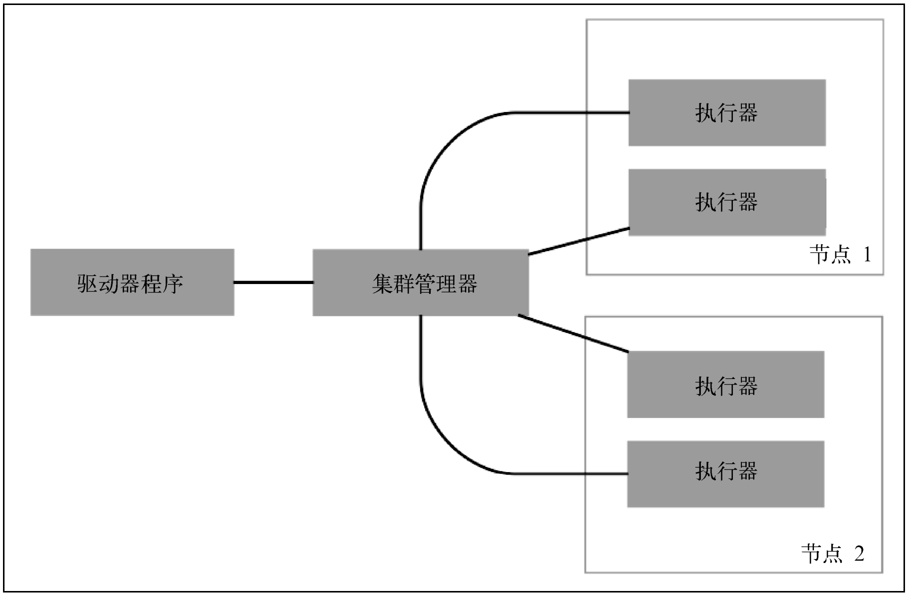
用户通过驱动器程序与集群管理器交互。负责在用户和 Spark 集群之间通信的类被称为 `SparkContext`，这个类能够根据用户可用的资源连接并配置集群上的执行器。

在大多数情况下，Spark 通过一种名为**弹性分布式数据集**（RDD）的数据结构来管理其数据。RDD 表示一个元素集合，它能够处理大型数据集，这是通过将数据集中的元素切片，再并行地操作这些切片实现的（请注意，几乎对用户隐藏了这种机制）。在合适的情况下，RDD 还可存储在内存中，以提高访问速度以及缓存访问开销极高的中间结果。

通过使用 RDD，可定义任务和变换（这很像在 Dask 中自动生成计算图），而在被请求时，集群管理器将自动将任务分派给空闲执行器去执行。

执行器接受集群管理器分配的任务，执行任务，并在需要的时候保留结果。请注意，执行器可能有多个内核，而集群中的每个节点都可能有多个执行器。一般而言，Spark 能够抵御执行器故障。

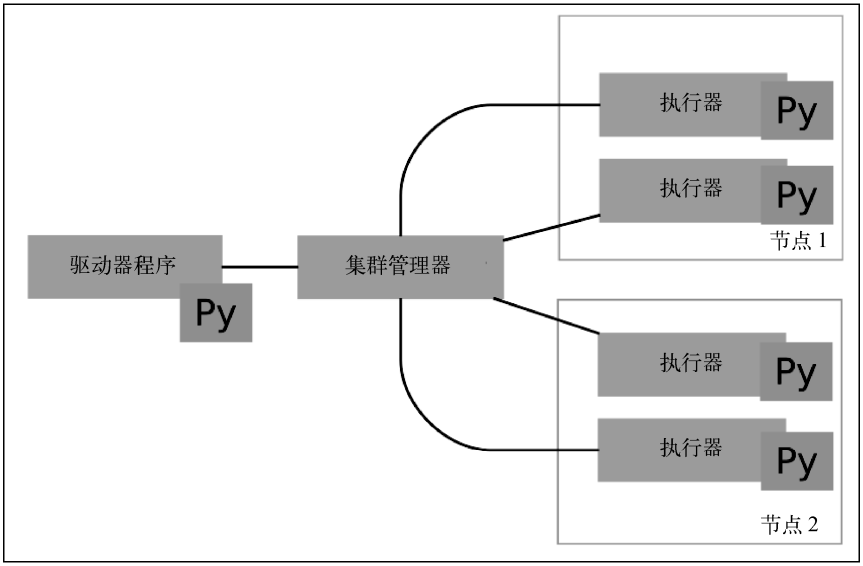
下图说明了 Spark 集群中前述组件是如何交互的。**驱动器程序**与**集群管理器**交互，而集群管理器管理不同节点上的**执行器**实例（每个执行器实例都可能有多个线程）。请注意，虽然**驱动器程序**不直接控制执行器，但存储在**执行器**实例上的结果将直接在执行器和驱动器程序之间传输，因此必须能够从**执行器**进程通过网络连接到**驱动器程序**。




那么问题来了，作为一款使用 Scala 编写的软件，Spark 怎么能够执行 Python 代码呢？集成是通过 Py4J 库实现的，这个库在幕后维护着一个 Python 进程，并通过套接字（一种进程间通信方式）与这个进程通信。为运行任务，执行器维护着一系列 Python 进程，以便并行地处理 Python 代码。

RDD 和在驱动器程序中的 Python 进程中定义的变量被串行化，而集群管理器和执行器之间的通信（包括 shuffling）是由 Spark 的 Scala 代码处理的。为了在 Python 和 Scala 之间交互，还必须执行额外的串行化步骤，这也将增加通信开销。因此，使用 PySpark 时必须特别小心，要确保使用的数据结构能够被高效地串行化，同时确保数据切片足够大，让通信开销相比于执行开销可以忽略不计。

下图列出了为执行 PySpark 所需的 Python 进程。这些多出来的 Python 进程会消耗内存，还增加了一个间接层，导致错误报告更加复杂。




虽然存在这些缺点，但 PySpark 依然被广泛使用，因为它在活跃的 Python 生态系统和行业领先的 Hadoop 基础设施之间架起了一座桥梁。

### 8.3.3　弹性分布式数据集

要在 Python 中创建 RDD，最简单的方式是使用方法 `SparkContext.parallelize`。在本章前面，我们使用这个方法并行化了一个包含整数 0~1000 的集合，如下所示。

```
rdd = sc.parallelize(range(1000))
# 结果：
# PythonRDD[3] at RDD at PythonRDD.scala:48
```

集合 `rdd` 将被分成很多个切片，这里为默认的 4 个（可使用配置选项来修改默认值）。要显式地指定切片个数，可向 `parallelize` 再传递一个参数。

```
rdd = sc.parallelize(range(1000), 2)
rdd.getNumPartitions() # 这个函数将返回切片个数
partitions
# 结果：
# 2
```

RDD 支持很多函数式编程运算符，就像第 6 章介绍的响应式编程和数据流（但在响应式编程中，运算符是设计用于处理事件而不是普通集合的）。我们来演示一下你现在应该很熟悉的基

本函数 map。在下面的示例中，我们使用了 map 来计算一系列数字的平方。

```
square_rdd = rdd.map(lambda x: x**2)
# 结果：
# PythonRDD[5] at RDD at PythonRDD.scala:48
```

函数 map 返回一个新的 RDD，而没有执行任何计算。要触发计算，可使用方法 collect，这将获取集合中的所有元素；也可使用方法 take，它只返回前 10 个元素。

```
square_rdd.collect()
# 结果：
# [0, 1, ... ]

square_rdd.take(10)
# 结果：
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

为比较 PySpark、Dask 和本书前面介绍过的其他并行编程库，我们将再次估算 *pi* 的值。在 PySpark 实现中，我们首先使用 parallelize 创建两个包含随机数的 RDD，再使用（与 Python 函数等效的）函数 zip 合并这两个数据集，然后检查这些随机点是否在圆内。

```
import numpy as np

N = 10000
x = np.random.uniform(-1, 1, N)
y = np.random.uniform(-1, 1, N)

rdd_x = sc.parallelize(x)
rdd_y = sc.parallelize(y)

hit_test = rdd_x.zip(rdd_y).map(lambda xy: xy[0] ** 2 + xy[1] ** 2 < 1)
pi = 4 * hit_test.sum()/N
```

必须指出的是，zip 和 map 操作生成新的 RDD，而不对底层数据执行指令。在刚才的示例中，将在我们调用函数 hit_test.sum 时触发代码执行，这个函数返回一个整数。这种行为不同于 Dask API，使用 Dask API 编写的所有代码（包括最终结果 pi）都不会触发代码执行。

下面通过一个更有趣的应用程序来演示其他的 RDD 方法。我们将学习如何计算网站的每位用户在一天内的访问次数。在实际编程中，数据已收集到数据库或存储在分布式文件系统（如 HDFS）中，但在这个示例中，我们将生成一些数据，再进行分析。

在下面的代码中，我们生成了一个字典列表，其中每个字典都包含一位用户（从 20 位用户中选出来的）和一个时间戳。生成这个数据集的步骤如下。



(1) 创建一个包含 20 位用户的用户池（变量 users）。

(2) 定义一个函数，返回一个介于两个日期之间的随机时间。

(3) 从用户池中随机选择一位用户，并随机选择一个介于 2017 年 1 月 1 日和 2017 年 1 月 7

日之间的时间。重复这种操作 10 000 次。

```
import datetime

from uuid import uuid4
from random import randrange, choice

# 生成 20 位用户
n_users = 20
users = [uuid4() for i in range(n_users)]

def random_time(start, end):
    '''返回一个介于起始日期和终止日期之间的随机时间戳'''
    # 选择一个用秒数表示的时间
    total_seconds = (end - start).total_seconds()
    return start +
    datetime.timedelta(seconds=randrange(total_seconds))

start = datetime.datetime(2017, 1, 1)
end = datetime.datetime(2017, 1, 7)

entries = []
N = 10000
for i in range(N):
    entries.append({
     'user': choice(users),
     'timestamp': random_time(start, end)
    })
```

生成数据集后，就可开始提问并使用 PySpark 来回答了。一个常见的问题是，某位用户访问了网站多少次。为回答这个问题，一种比较幼稚的办法是，将 RDD 中的条目按用户分组（使用运算符 `groupBy`），并计算每位用户有多少个条目。在 PySpark 中，`groupBy` 将一个用于提取分组键的函数作为参数，并返回一个新的 RDD，其中包含形如 `(key, group)` 的元组。在下面的示例中，我们将用户 ID 作为键提供给 `groupBy`，并使用 `first` 来查看第一个元素。

```
entries_rdd = sc.parallelize(entries)
entries_rdd.groupBy(lambda x: x['user']).first()
# 结果：
# (UUID('0604aab5-c7ba-4d5b-b1e0-16091052fb11'),
#  <pyspark.resultiterable.ResultIterable at 0x7faced4cd0b8>)
```

在 `groupBy` 返回的值中，每个用户 ID 都有一个对应的 `ResultIterable`（大致相当于一个列表）。要计算每位用户的访问次数，只需计算每个 `ResultIterable` 的长度即可。

```
(entries_rdd
 .groupBy(lambda x: x['user'])
 .map(lambda kv: (kv[0], len(kv[1])))
 .take(5))
# 结果：
# [(UUID('0604aab5-c7ba-4d5b-b1e0-16091052fb11'), 536),
#  (UUID('d72c81c1-83f9-4b3c-a21a-788736c9b2ea'), 504),
```

```
#  (UUID('e2e125fa-8984-4a9a-9ca1-b0620b113cdb'), 498),
#  (UUID('b90acaf9-f279-430d-854f-5df74432dd52'), 561),
#  (UUID('00d7be53-22c3-43cf-ace7-974689e9d54b'), 466)]
```

对于小型数据集来说，这个算法的效果很好，但 groupBy 要求收集每位用户的所有条目并将其存储到内存中，而这可能会超过节点的内存量。由于我们不需要这个列表，而只需要访问次数，因此一种更佳的方式是，只计算每位用户的访问次数，而不将其访问列表存储到内存中。

TIP 处理由键–值对组成的 RDD 时，可使用 mapValues 将一个函数应用于值。在前面的代码中，可将调用 map(lambda kv: (kv[0], len(kv[1])))替换为 mapValues(len)，这样可读性更好。

为提高计算效率，可使用函数 reduceByKey，它执行的操作类似于本章前面“MapReduce 简介”一节介绍的 Reduce 步骤。前面的 RDD 由元组组成，其中每个元组的第一个元素都是键，而第二个元素为值，因此对其调用函数 reduceByKey，并将一个执行归并计算的函数作为其第一个参数。下面的代码演示了函数 reduceByKey 的一种简单用法。在这个示例中，有一些与整数相关联的字符串键，我们要获取相同键关联的所有值的和，这是使用 lambda 表达式表示的归并函数实现的。

```
    rdd = sc.parallelize([("a", 1), ("b", 2), ("a", 3), ("b", 4), ("c",
5)])
    rdd.reduceByKey(lambda a, b: a + b).collect()
    # 结果：
    # [('c', 5), ('b', 6), ('a', 4)]
```

函数 reduceByKey 的效率比 groupBy 高得多，因为归并操作是可并行化的，同时不需要在内存中存储分组。另外，它还避免了在执行器之间传输数据（它执行的操作与本章前面介绍的 Dask 运算符 foldby 类似）。现在可以使用 reduceByKey 重写计算访问次数的代码了。

```
(entries_rdd
 .map(lambda x: (x['user'], 1))
 .reduceByKey(lambda a, b: a + b)
 .take(3))
# 结果：
# [(UUID('0604aab5-c7ba-4d5b-b1e0-16091052fb11'), 536),
#  (UUID('d72c81c1-83f9-4b3c-a21a-788736c9b2ea'), 504),
#  (UUID('e2e125fa-8984-4a9a-9ca1-b0620b113cdb'), 498)]
```

使用 Spark 的 RDD API，还可轻松地回答下面这样的问题：网站在每天中都被访问了多少次？这可使用 reduceByKey 和合适的键（从时间戳中提取的日期）来计算。下面的示例演示了如何计算。另外，还使用了运算符 sortByKey 将返回的访问次数按日期排序。

```
(entries_rdd
 .map(lambda x: (x['timestamp'].date(), 1))
 .reduceByKey(lambda a, b: a + b)
 .sortByKey()
 .collect())
```

```
# 结果：
# [(datetime.date(2017, 1, 1), 1685),
#  (datetime.date(2017, 1, 2), 1625),
#  (datetime.date(2017, 1, 3), 1663),
#  (datetime.date(2017, 1, 4), 1643),
#  (datetime.date(2017, 1, 5), 1731),
#  (datetime.date(2017, 1, 6), 1653)]
```

### 8.3.4　Spark **DataFrame**

对于数值计算和分析任务，Spark 通过模块 `pyspark.sql`（也叫 SparkSQL）提供了一个便利的接口。这个模块包含一个 `spark.sql.DataFrame` 类，可用来高效地执行 SQL 式查询，就像 Pandas 中那样。要访问 SQL 接口，可创建一个 `SparkSession` 对象。

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

然后，通过这个对象调用函数 `createDataFrame` 来创建一个 `DataFrame`，这个函数将一个 RDD、列表或 `pandas.DataFrame` 作为参数。

在下面的示例中，我们通过转换一个包含一系列 `Row` 实例的 RDD（`rows`）来创建一个 `spark.sql.DataFrame`。`Row` 实例就像 `pd.DataFrame` 中的行一样，将一组列名关联到一组值。在这个示例中，有两列——`x` 和 `y`，我们将它们关联到随机数。

```
# 使用前面定义的x_rdd和y_rdd
rows = rdd_x.zip(rdd_y).map(lambda xy: Row(x=float(xy[0]), y=float(xy[1])))

rows.first() # 查看第一个元素
# 结果：
# Row(x=0.18432163061239137, y=0.632310101419016)
```

有了 `Row` 实例集合后，就可将它们合并成一个 `DataFrame`，如下所示。我们还可使用方法 `show` 查看这个 `DataFrame` 的内容。

```
df = spark.createDataFrame(rows)
df.show(5)
# 输出：
# +-------------------+--------------------+
# |                  x|                   y|
# +-------------------+--------------------+
# |0.18432163061239137|   0.632310101419016|
# | 0.8159145525577987| -0.9578448778029829|
# |-0.6565050226033042|  0.4644773453129496|
# |-0.1566191476553318|-0.11542211978216432|
# | 0.7536730082381564| 0.26953055476074717|
# +-------------------+--------------------+
# 只显示了前5行
```

spark.sql.DataFrame 支持使用便利的 SQL 语法对分布式数据集执行变换。例如，可使用方法 selectExpr 来计算 SQL 表达式的值。在下面的代码中，我们使用了 x 和 y 列以及 SQL 函数 pow 来检查是否击中。

```
hits_df = df.selectExpr("pow(x, 2) + pow(y, 2) < 1 as hits")
hits_df.show(5)
# 输出：
# +-----+
# | hits|
# +-----+
# | true|
# |false|
# | true|
# | true|
# | true|
# +-----+
# 只显示了前 5 行
```

为了演示 SQL 强大的表达力，我们还可使用一个表达式来估算 *pi* 的值。在这个表达式中，使用了 sum、pow、cast 和 count 等 SQL 函数。

```
result = df.selectExpr('4 * sum(cast(pow(x, 2) +
                        pow(y, 2) < 1 as int))/count(x) as pi')
result.first()
# 结果：
# Row(pi=3.13976)
```

Spark SQL 的语法与 Hive 相同。Hive 是一个建立在 Hadoop 基础之上的分布式数据集 SQL 引擎。要全面了解其语法，请参阅 https://cwiki.apache.org/confluence/display/Hive/LanguageManual。

要通过 Python 接口利用 Scala 的威力及其所做的优化，DataFrame 是绝佳的途径，其中的主要原因是，虽然查询在名义上是由 SparkSQL 解释的，但实际上是直接在 Scala 中执行的，中间结果不会经过 Python。这极大地降低了串行化开销，并利用了 SparkSQL 所做的查询优化。优化和查询规划让你能够使用 SQL 运算符，如 GROUP BY，同时不会像直接对 RDD 使用 groupBy 那样降低性能。

## 8.4　使用 mpi4py 执行科学计算

虽然 Dask 和 Spark 是很出色的技术，在 IT 行业得到了广泛使用，但在学术研究领域还未被广泛采纳。在学术界，几十年来一直使用包含数千个处理器的超级计算机来运行执行大量数值计算的应用程序，因此通常超级计算机运行的软件截然不同，这些软件专注于使用 C、Fortran 乃至汇编语言等低级语言实现计算密集型算法。

在这种系统上，用来实现并行执行的主要库是**消息传递接口**（MPI），这个接口虽然不像 Dask

和 Spark 那样便利和精致，但完全能够表达并行算法并获得极佳的性能。请注意，不同于 Dask 和 Spark，MPI 并没有采用 MapReduce 模型，因此最适合用来运行数千个几乎不相互发送数据的进程。

MPI 的工作原理与本章前面介绍的截然不同。在 MPI 中，并行性是通过在多个可能位于不同节点上的进程中运行**同一个脚本**实现的；进程之间的通信和同步由一个专门的进程处理，这个进程通常被称为**根**（root），并由 ID 0 标识。

在本节中，我们将以 mpi4py（Python MPI 接口）为例，初略地演示主要的 MPI 概念。下面的示例演示了使用 MPI 编写的最简单的代码。这些代码导入模块 MPI，并获取 COMM_WORLD——一个可用来与其他 MPI 进程交互的接口。函数 Get_rank 返回当前进程的整型标识符：

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
print("This is process", rank)
```

我们可将这些代码放在文件 mpi_example.py 中，并执行这个文件。运行这个脚本通常不会做任何特殊的事情，因为它只在一个进程中执行。

```
$ python mpi_example.py
This is process 0
```

MPI 作业应该使用命令 mpiexec 来执行，这个命令包含指定并行进程数的选项-n。使用下面的命令运行这个脚本将生成 4 个不同的进程，它们执行同一个脚本，ID 各不相同。

```
$ mpiexec -n 4 python mpi_example.py
This is process 0
This is process 2
This is process 1
This is process 3
```

通过使用资源管理器（如 TORQUE），进程将自动分散到网络中。通常，超级计算机都是由系统管理员配置的，他们会提供有关如何运行 MPI 软件的说明。

为了让你感觉一下 MPI 程序是什么样的，我们再次来实现 *pi* 值估算。完整的代码如下面所示。这个程序所做的工作如下。

- 为每个进程创建一个长度为 N / n_procs 的随机数组，让每个进程检查相同数量的样本（n_procs 是使用函数 Get_size 获得的）。
- 在每个进程中，计算击中检查结果之和，并将其存储在 hits_counts 中，它表示每个进程检查到的击中次数。
- 使用函数 reduce 计算所有进程检查到的击中次数之和。使用 reduce 时，需要指定使用参数 root 来指定哪个进程将收到结果。

❑ 只在根进程中打印最终结果：

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

import numpy as np

N = 10000

n_procs = comm.Get_size()

print("This is process", rank)

# 创建一个数组
x_part = np.random.uniform(-1, 1, int(N/n_procs))
y_part = np.random.uniform(-1, 1, int(N/n_procs))

hits_part = x_part**2 + y_part**2 < 1
hits_count = hits_part.sum()

print("partial counts", hits_count)

total_counts = comm.reduce(hits_count, root=0)

if rank == 0:
   print("Total hits:", total_counts)
   print("Final result:", 4 * total_counts/N)
```

现在可以将上述代码放在文件 mpi_pi.py 中，并使用 `mpiexec` 来执行这个文件。输出表明，在 `reduce` 调用前，四个进程同时执行。

```
$ mpiexec -n 4 python mpi_pi.py
This is process 3
partial counts 1966
This is process 1
partial counts 1944
This is process 2
partial counts 1998
This is process 0
partial counts 1950
Total hits: 7858
Final result: 3.1432
```

## 8.5　小结

分布式处理可通过在计算机集群中分配小型任务，实现能够处理超大数据集的算法。多年来，为实现性能卓越而又可靠的分布式软件，开发出了 Apache Hadoop 等众多软件包。

本章介绍了 Dask 和 PySpark 等 Python 包的架构和用法，它们提供了功能强大的 API，让你能够设计可在数百台计算机上运行的程序。我们还简要地介绍了 MPI 库，几十年来它一直用于在超级计算机（用于学术研究）上分配工作。

到目前为止，本书探索了多种程序性能改进方法，使用这些方法可提高程序的速度，并使其能够处理更大的数据集。下一章将介绍编写和维护高性能代码的策略和最佳实践。

# 第 9 章 高性能设计

在前几章，我们学习了如何使用 Python 标准库和第三方包中的各种工具，来评估和改善 Python 应用程序的性能。本章将提供有关如何设计各种应用程序的一般性指南，并演示一些被多个 Python 项目采用的最佳实践。

本章介绍如下主题：

- 为普通应用程序、数值计算应用程序和大数据应用程序选择合适的性能优化策略；
- 组织 Python 项目；
- 使用虚拟环境和容器隔离 Python；
- 使用 Travis CI 实现持续集成。

## 9.1 选择合适的策略

可用于改善程序性能的包很多，但如何确定程序的最佳优化策略呢？该使用哪种优化方式取决于很多因素，本章将力图基于应用程序的类型尽可能全面地回答这个问题。

首先要考虑的是应用程序的类型。Python 语言被用于众多不同的领域，包括 Web 服务、系统脚本、游戏、机器学习等。对于不同的应用程序，需要优化的部分也不同。

例如，对于 Web 服务，可通过优化使其响应时间极短；它还必须能够处理尽可能多的请求，同时使用尽可能少的资源（即尽可能缩短延迟）。而数值计算代码可能需要几周才能运行完毕，因此提高系统能够处理的数据量很重要，即便启动开销很大也无妨（在这种情况下，我们在乎的是吞吐量）。

另一个方面是开发的应用程序要在什么平台和体系结构中运行。Python 支持很多平台和体系结构，但很多第三方库对有些平台的支持可能有限，尤其是涉及 C 语言扩展的包。因此，必须核实原本打算使用的库是否可用于目标平台和体系结构。

另外，有些体系结构（如嵌入式系统和小型设备）的 CPU 处理能力和内存可能有限。这是

一个必须考虑的重要因素，因为有些技术（如多处理）可能会消耗太多内存，或者要求执行额外的软件。

最后，业务需求也同样重要。在很多情况下，软件产品必须快速迭代，并能够快速修改其代码。一般而言，你希望软件栈尽可能小，这样才可能在短时间内完成修改、测试和部署以及添加对其他平台的支持。这也适用于团队开发——安装软件栈和为开发做好准备的工作应尽可能容易。有鉴于此，通常应选择使用纯粹的 Python 库，而不是扩展，但久经考验的库（如 NumPy）可能例外。另外，很多业务方面决定了应首先优化哪些操作（千万不要忘了，**过早优化是万恶之源**）。

### 9.1.1　普通应用程序

Web 应用和移动应用后端等普通应用程序，通常需要调用远程服务和数据库。在这种情况下，使用异步框架（如第 6 章介绍的框架）可能大有裨益，因为这将改善应用程序的逻辑、系统设计、响应速度等，还将简化网络故障的处理工作。

使用异步编程还让微服务实现和使用起来更容易。**微服务**虽然没有权威的定义，但可将其视为专注于应用程序某个方面的远程服务，如身份验证。

微服务背后的理念是，可将通过简单协议（如 gRPC 和 REST 调用，或专用消息队列）进行通信的微服务组合起来，从而打造出应用程序。这种体系结构与单体应用程序完全不同。在单体应用程序中，所有的服务都是由同一个 Python 进程处理的。

微服务的优点之一在于，应用程序的不同部分完全解耦。简单的小型服务可由不同的团队实现和维护，还可在不同的时间进行更新和部署。这样就能够轻松地复制微服务，以便处理更多的用户。另外，由于通信是通过简单协议进行的，因此可使用比 Python 更合适的语言来实现微服务。

如果对服务的性能不满意，通常可在不同的 Python 解释器（如 PyPy）上执行应用程序（条件是所有第三方扩展都是兼容的），以获得足够的速度提升。如果这样做不可行，通过调整算法策略并将瓶颈部分移植到 Cython，通常足以获得满意的性能。

### 9.1.2　数值计算代码

如果你要编写的是数值计算代码，一种极好的策略是一开始就使用 NumPy。使用 NumPy 是一种稳妥的选择，因为它可用于很多平台并久经考验，而且正如你在本书前面看到的，很多其他的包都将 NumPy 数组视为一等公民。

只要妥善地编写（如使用第 2 章介绍的广播等技术），NumPy 的性能几乎能够与 C 代码的性

能媲美，无须进一步优化。虽然如此，有些算法使用 NumPy 数据结构和方法难以高效地表示。在这种情况下，两个很不错的选择是 Numba 或 Cython。

Cython 是一个非常成熟的工具，被很多重要的项目广泛采用，如 `scipy` 和 `scikit-learn`。Cython 代码包含显式的静态类型声明，因此很容易理解，大多数 Python 程序员都能学会其语法。另外，神奇而良好的查看工具让程序员能够轻松地预测性能，并就怎样修改将最大限度地提高性能做出有根据的猜测。

然而，Cython 也有一些缺点。执行 Cython 代码前必须编译，这破坏了 Python 便利的编辑–运行周期。这还要求必须有用于目标平台的兼容 C 编译器。另外，这还导致分发和部署工作更复杂，因为需要测试多个平台、体系结构、配置和编译器。

另一方面，Numba API 只要求定义纯粹的 Python 函数，而这些函数将被动态地编译，从而保留了 Python 快速的编辑–运行周期。一般而言，Numba 要求目标平台安装了 LLVM 工具链。请注意，在 0.30 版中，对**预先**（AOT）编译 Numba 函数提供了一定的支持，因此可打包并部署编译好的 Numba 函数，这样就不用在目标平台上安装 Numba 和 LLVM。

请注意，包管理器 `conda` 以打好包的方式提供 Numba 和 Cython，包中包含所有的依赖（包括编译器），因此在可使用包管理器 `conda` 的平台上，部署 Cython 的工作得以极大地简化。

> 如果 Cython 和 Numba 无法胜任，还可采取另一种策略：实现一个纯粹的 C 语言模块（这种模块可使用编译器标志或手工调整进一步优化），并在 Python 模块中通过 `cffi` 包或 Cython 来使用它，但通常不需要这样做。

使用 NumPy、Numba 和 Cython 是非常有效的策略，对于串行代码，几乎可获得最优的性能。对很多应用程序来说，使用串行代码就足够了，即便最终决定使用并行算法，开发出串行参考实现也是非常值得的，这样可方便调试，因为在数据集较小的情况下，串行实现的速度通常更快。

并行实现的复杂性随应用程序的不同差别很大。在很多情况下，对于可轻松地表示为一系列独立计算和某种**聚合**的程序，可使用基于进程的简单接口（如 `multiprocessing.Pool` 或 `ProcessPoolExecutor`）进行并行化，这些接口的优点是不用费多大力气就能并行地执行普通 Python 代码。

为避免启动多个进程的时间和内存开销，可使用线程。NumPy 函数通常会释放 GIL，因此非常适合采用基于线程的并行化。另外，Cython 和 Numba 提供了特殊的 `nogil` 语句和并行自动化，因此它们适合用于简单的轻量级并行化。

对于更复杂的情况，可能必须大刀阔斧地修改算法。在这种情况下，Dask 数组是不错的选择，它几乎是标准 NumPy 的简单替代品。Dask 还有另一个优点，那就是其操作非常透明，因此很容易调整。

大量使用线性代数例程的专用应用程序（如深度学习和计算机图形学应用程序）可能受益于 Theano 和 Tensorflow 等包，这些包性能卓越，能够自动并行化且内置了 GPU 支持。

最后，要将并行的 Python 脚本部署到基于 MPI 的超级计算机（通常供高校的研究人员使用），可使用 `mpi4py` 包。

### 9.1.3　大数据

大型数据集（通常超过 1TB）日益普遍，目前已为开发能够收集、存储和分析这种数据集的技术，投入了大量的资源。通常，根据这些数据原本是如何存储的来决定选择使用哪种框架。

在很多情况下，单台计算机无法存储整个数据集，但通过采取合适的策略，无须研究整个数据集就能找到问题的答案。例如，可提取一小部分感兴趣的数据（这些数据可轻松地加载到内存中），再使用方便而出色的库（如 Pandas）进行分析，这样很可能能够回答问题。对于业务问题，通过筛选或随机采集数据点，通常可找到足够准确的答案，而无须求助于大数据工具。

如果公司的大部分软件都是使用 Python 编写的，且你对使用什么样的软件栈有决定权，则使用 Dask distributed 是个不错的选择。这个软件包安装起来非常简单，且与 Python 生态系统集成紧密。使用 Dask 数组和 `DataFrame` 时，很容易通过修改 NumPy 和 Pandas 代码来改善既有 Python 算法的性能。

如果公司已搭建了 Spark 集群，PySpark 将是最佳的选择。如果要进一步提高性能，可使用 SparkSQL。Spark 的优点之一是，允许你使用其他的语言，如 Scala 和 Java。

## 9.2　组织代码

典型 Python 项目的仓库结构至少包含一个目录，这个目录包含如下内容：文件 README.md；一个 Python 模块或包，其中包含应用程序或库的源代码；一个 setup.py 文件。项目还可能遵循其他约定，以便符合公司的策略或使用的框架的要求。本节将介绍社区驱动的 Python 项目（包括本书前面介绍的一些工具）常采取的一些做法。

下面是一个名为 myapp 的 Python 项目的典型目录结构：

```
myapp/
  README.md
  LICENSE
  setup.py
  myapp/
    __init__.py
    module1.py
    cmodule1.pyx
    module2/
```

```
    __init__.py
src/
  module.c
  module.h
tests/
  __init__.py
  test_module1.py
  test_module2.py
benchmarks/
  __init__.py
  test_module1.py
  test_module2.py
docs/
tools/
```

下面来详细说说其中的每个文件和目录。

README.md 是一个文本文件，包含有关软件的一般性信息，如项目范围、安装方法、简明教程和有用的链接。如果软件是公开发行的，还有一个 LICENSE 文件，其中包含使用条款和条件。

通过使用 `setuptools` 库将 Python 软件打包到一个名为 setup.py 的文件中。正如你在本书前面看到的，setup.py 也是一种编译并分发 Cython 代码的有效方式。

`myapp` 包包含应用程序的源代码，其中包括 Cython 模块。在有些情况下，除优化的 Cython 实现外，保留纯粹的 Python 实现可提供便利。通常，Cython 版模块的名称以字母 c 打头（如上述示例中的 `cmodule1.pyx`）。

如果需要外部.c 和.h 文件，这些文件通常存储在项目顶级目录（myapp）下的目录 src/中。

目录 tests/包含应用程序的测试代码（通常为单元测试），可使用测试运行器（如 `unittest` 或 `pytest`）来运行它们。然而，有些项目选择将目录 tests/放在 `myapp` 包中。由于高性能代码需要反复调整和重写，必须有可靠的测试套件，这样才能尽早发现 bug，并缩短测试–编辑–运行周期，进而改善开发体验。

基准测试程序可放在目录 benchmarks 中。基准测试程序的执行时间可能很长，通过将基准测试程序与测试分开，可避免测试时间太长。也可在构建服务器（参见 9.4 节）上运行基准测试程序，将此作为一个比较不同版本性能的简单方式。虽然基准测试程序的运行时间通常比单元测试长，但最好让其执行时间尽可能短，以免浪费资源。

最后，目录 docs/包含用户和开发文档以及 API 参考，通常还包含文档工具（如 `sphinx`）的配置文件。其他工具和脚本可放在目录 tools/中。

## 9.3 隔离、虚拟环境和容器

在隔离环境中测试和执行代码很重要，因为这样才能准确地告诉朋友如何运行你的 Python 脚本：安装 Python X 版以及依赖包 Y 和 X，再将脚本复制到计算机中并执行它。

在很多情况下，你的朋友会去下载用于其平台的 Python 和依赖库，再尝试执行脚本。然而，脚本很可能运行失败，因为朋友的计算机安装的操作系统与你的不同，或者他安装的库版本与你安装的不同，还可能他以前安装的库没有妥善地删除，导致难以发现的冲突和很多麻烦。

为避免这种情况发生，一种非常简单的办法是使用虚拟环境。虚拟环境将 Python、相关的可执行文件和第三方包隔离，让你能够创建和管理多个 Python 安装。从 Python 3.3 起，标准库包含模块 `venv`（以前名为 `virtualenv`），这是一个设计用于创建和管理隔离环境的工具。在基于 `venv` 的虚拟环境中，可使用 setup.py 文件或 `pip` 来安装 Python 包。

对于高性能代码，准确而详细地指出使用的是哪个版本的库至关重要。库会随新版本的推出而发展，而算法的变换将极大地影响性能。例如，`scipy` 和 `scikit-learn` 等流行的库经常将其代码和数据结构移植到 Cython，因此要获得最佳的性能，用户必须安装正确版本的库。

### 9.3.1 使用 conda 环境

在大多数情况下，使用 `venv` 就挺好，但编写高性能代码时，经常会遇到这样的情况：有些高性能库要求安装非 Python 软件。这通常要求进一步设置编译器以及 Python 包链接的高性能原生库（它们是使用 C、C++或 Fortran 编写的）。由于 `venv` 和 `pip` 只能处理 Python 包，因此这些工具无法处理这样的情况。

包管理器 `conda` 是专门为应对这种情形而创建的。要使用 `conda` 创建虚拟环境，可使用命令 `conda create`。这个命令接受一个`-n` 参数（`-n` 表示`--name`，给新创建的环境指定标识符）以及要安装的包。例如，要创建一个使用 Python 3.5 和最新版 NumPy 的环境，可使用如下命令：

```
$ conda create -n myenv Python=3.5 numpy
```

`conda` 会负责从其仓库中获取相关的包，并将它们放在一个隔离的 Python 安装中。要启用虚拟环境，可使用命令 `source activate`。

```
$ source activate myenv
```

执行这个命令后，默认的 Python 解释器将设置为前面指定的版本。要核实 Python 可执行文件的位置，可使用命令 `which`，它返回这个可执行文件的完整路径。

```
(myenv) $ which python
/home/gabriele/anaconda/envs/myenv/bin/python
```

现在，你可在虚拟环境中随便添加、删除和修改包，而不会影响全局 Python 安装。要安装其他的包，可使用命令 `conda install <package name>`，也可使用 `pip`。

虚拟环境的优点在于，你能够以隔离的方式安装和编译任何软件。这意味着如果虚拟环境因某种原因受损，你可推倒重来。

要删除虚拟环境 `myenv`，需要先禁用它，再使用命令 `conda env remove`，如下所示。

```
(myenv) $ source deactivate
$ conda env remove -n myenv
```

如果标准 `conda` 仓库中没有要安装的包，该怎么办呢？一种选择是看看社区频道 `conda-forge` 有没有。要在 `conda-forge` 中搜索包，可使用命令 `conda search` 并指定选项 `-c`（表示`--channel`）。

```
$ conda search -c conda-forge scipy
```

这个命令将列出一系列与查询字符串 `scipy` 匹配的包及其版本。另一种选择是在 Anaconda Cloud 上托管的公共频道中搜索。要下载 Anaconda Cloud 命令行客户端，可安装 `anaconda-client` 包。

```
$ conda install anaconda-client
```

安装命令行客户端 `anaconda` 后，就可使用它来搜索包了。下面的示例演示了如何查找 `chemview` 包。

```
$ anaconda search chemview
Using Anaconda API: https://api.anaconda.org
Run 'anaconda show <USER/PACKAGE>' to get more details:
Packages:
 Name                         | Version | Package Types   | Platforms
 ------------------------- | ------  | --------------- | ---------------
 cjs14/chemview               | 0.3     | conda           | linux-64, win-64,
osx-64
                                        : WebGL Molecular Viewer for IPython
notebook.
 gabrielelanaro/chemview      | 0.7     | conda           | linux-64, osx-64
                                        : WebGL Molecular Viewer for IPython
notebook.
```

然后就可通过指定合适的频道和选项`-c`，轻松进行安装了。

```
$ conda install -c gabrielelanaro chemlab
```

### 9.3.2 虚拟化和容器

很久以前，虚拟化就已面世，它让你能够在同一台计算机中运行多个操作系统，以更好地利

用物理资源。

要实现虚拟化，一种方式是使用**虚拟机**。虚拟机创建虚拟硬件资源，如 CPU、内存和设备，并且使用它们在同一台计算机上安装并运行多个操作系统。要实现虚拟化，可在一个操作系统上安装 hypervisor 应用程序。这个操作系统被称为**宿主**（host）。hypervisor 能够创建、管理和监视虚拟机及其操作系统（被称为**来宾**，guest）。

需要指出的是，虚拟环境虽然包含“虚拟”二字，但与虚拟机没有任何关系。虚拟环境是 Python 特有的，通过 shell 脚本来设置不同的 Python 解释器。

容器是一种隔离应用程序的方式，它创建一个独立于宿主操作系统的环境，其中只包含必要的依赖。容器是一个操作系统特性，让你能够在多个实例之间共享操作系统内核提供的硬件资源。容器不同于虚拟机，因为它不抽象硬件资源，而只分享操作系统内核。

在利用硬件方面，容器的效率极高，因为它通过内核以原生方式访问硬件。因此，对高性能应用程序来说，容器是极佳的解决方案。容器还可快速地创建和删除，可用于以隔离的方式快速测试应用程序。容器还可用来简化部署工作（尤其是微服务），以及开发构建服务器，如前一节提到的构建服务器。

在第 8 章，我们使用 Docker 轻松地搭建了一个 PySpark 环境。Docker 是当前最受欢迎的容器化解决方案之一。要安装 Docker，最佳的方式是按官网上的说明操作。安装后就可轻松地使用其命令行界面来创建和管理容器。

要启动一个新容器，可使用命令 `docker run`。在接下来的示例中，我们将演示如何使用 `docker run` 在一个 Ubuntu 16.04 容器中执行 shell 会话。为此，需要指定如下参数。

- `-i` 指定我们要启动一个交互式会话。也可以非交互方式执行 `docker` 命令（如启动 Web 服务器）。
- `-t <image name>`指定要使用哪个系统镜像。在下面的示例中，我们使用的是镜像 `ubuntu:16.04`。
- `/bin/bash` 是要在容器中运行的命令，如下所示。

```
$ docker run -i -t ubuntu:16.04 /bin/bash
root@585f53e77ce9:/#
```

这将命令将立即带我们进入一个隔离的 shell，我们可在其中把玩系统和安装软件，而不会影响宿主操作系统。要在不同的 Linux 版本中测试安装和部署，使用容器是一种极佳的方式。使用完这个交互式 shell 后，可执行命令 `exit` 返回宿主系统。

在前一章运行可执行文件 pyspark 时，我们还使用了分离选项`-d` 和端口选项`-p`。选项`-d` 只是让 Docker 在后台运行命令。选项`-p <host_port>:<guest_port>`是必不可少的，它将宿主

操作系统的一个网络端口映射到来宾系统；如果没有这个选项，在宿主系统中运行的浏览器将无法访问 Jupyter notebook。

要监视容器的状态，可使用命令 docker ps，如下面的代码所示。选项-a（表示 all）指定输出所有容器的信息，而不管它们当前是否在运行。

```
$ docker ps -a
CONTAINER ID IMAGE        COMMAND     CREATED        STATUS     PORTS NAMES
585f53e77ce9 ubuntu:16.04 "/bin/bash" 2 minutes ago Exited (0)       2
minutes ago pensive_hamilton
```

docker ps 提供的信息包括一个用十六进制数表示的标识符（585f53e77ce9），还有方便人类阅读的容器名（pensive_hamilton）。在其他 docker 命令中，它们都可用来指定容器。输出中还包含其他信息，如执行的命令、创建时间以及容器的当前状态。

要恢复执行已退出的容器，可使用命令 docker start。要让容器访问 shell，可使用命令 docker attach。在这两个命令中，可使用 ID 来指定容器，也可使用名称来指定容器。

```
$ docker start pensive_hamilton
pensive_hamilton
$ docker attach pensive_hamilton
root@585f53e77ce9:/#
```

要删除容器很容易，只需使用命令 docker rm 并指定容器的标识符即可。

```
$ docker rm pensive_hamilton
```

如你所见，你可随便执行命令，运行、停止和恢复容器，完成这些操作所需的时间都不超过 1 秒钟。要测试代码和尝试使用新包，同时又不影响宿主操作系统，交互地使用 Docker 容器是一种绝佳的方式。由于可同时运行很多容器，Docker 还可用来模拟分布式系统（以便进行测试和学习），而不要求有昂贵的计算集群。

Docker 还让你能够创建自己的系统镜像，这对分发、测试、部署和编写文档很有用。下一小节将介绍这个主题。

### 创建 Docker 镜像

Docker 镜像是预先配置好的可直接使用的系统。DockerHub 是一个 Web 服务，Docker 包的维护者将可直接使用的镜像上传到这里，供你用来测试和部署各种应用程序。要访问并安装 DockerHub 提供的 Docker 镜像，可使用命令 docker run。

要创建 Docker 镜像，一种方式是对既有容器执行命令 docker commit。这个命令将一个容器引用和输出镜像名称作为参数。

```
$ docker commit <container_id> <new_image_name>
```

9

要保存容器的快照时，这种方法很有用，但将镜像从系统中删除后，重新创建镜像的操作步骤也将丢失。

一种更佳的镜像创建方式是使用 Dockerfile。Dockerfile 是一个文本文件，提供了从另一个镜像开始构建新镜像的指令。下面来看看前一章中用来搭建支持 Jupyter notebook 的 PySpark 环境的 Dockerfile 的内容。

每个 Dockerfile 都需要一个起始镜像，这可使用命令 `FROM` 来指定。在这个示例中，起始镜像为 `jupyter/scipy-notebook`，这可从 DockerHub 获得。

指定起始镜像后，就可开始使用一系列 `RUN` 和 `ENV` 命令来执行 shell 命令，以安装包以及执行其他配置。在下面的示例中，安装了 Java 运行时环境（`openjdk-7-jre-headless`）、下载了 Spark 并设置了相关的环境变量。要指定接下来的命令由哪个用户执行，可使用 `USER` 指令。

```
FROM jupyter/scipy-notebook
MAINTAINER Jupyter Project <jupyter@googlegroups.com>
USER root

# Spark 依赖
ENV APACHE_SPARK_VERSION 2.0.2
RUN apt-get -y update &&
    apt-get install -y --no-install-recommends
    openjdk-7-jre-headless &&
    apt-get clean &&
    rm -rf /var/lib/apt/lists/*
RUN cd /tmp &&
    wget -q http://d3kbcqa49mib13.cloudfront.net/spark-
    ${APACHE_SPARK_VERSION}-bin-hadoop2.6.tgz    &&
    echo "ca39ac3edd216a4d568b316c3af00199
          b77a52d05ecf4f9698da2bae37be998a
          *spark-${APACHE_SPARK_VERSION}-bin-hadoop2.6.tgz" |
    sha256sum -c - &&
    tar xzf spark-${APACHE_SPARK_VERSION}
    -bin-hadoop2.6.tgz -C /usr/local &&
    rm spark-${APACHE_SPARK_VERSION}-bin-hadoop2.6.tgz
RUN cd /usr/local && ln -s spark-${APACHE_SPARK_VERSION}
    -bin-hadoop2.6 spark

# Spark 和 Mesos 配置
ENV SPARK_HOME /usr/local/spark
ENV PYTHONPATH $SPARK_HOME/python:$SPARK_HOME/python/lib/
    py4j-0.10.3-src.zip
ENV SPARK_OPTS --driver-java-options=-Xms1024M
    --driver-java-options=-
    Xmx4096M --driver-java-options=-Dlog4j.logLevel=info

USER $NB_USER
```

要使用 Dockerfile 来创建镜像，可切换到 Dockerfile 所在的目录，并执行下面的命令。可使用

选项-t 来指定用于存储镜像的标签( tag )。下面的命令使用前面的 Dockerfile 创建一个名为 pyspark 的镜像。

```
$ docker build -t pyspark .
```

这个命令将自动获取起始镜像 jupyter/scipy-notebook，并生成一个名为 pyspark 的新镜像。

## 9.4　持续集成

为确保应用程序中在每个开发迭代中都没有 bug，持续集成是一种绝佳方式。持续集成背后的主要理念是，非常频繁地运行项目的测试套件，这通常是在一台独立的构建服务器上进行的，该服务器直接从主项目仓库获取（pull）代码。

要搭建构建服务器，可在一台计算机上手动安装 Jenkins、Buildbot、Drone 等软件。这是一种便利且价格低廉的解决方案，对小型团队和私有项目来说尤其如此。

大多数开源项目都使用 Travis CI，这个服务能够自动使用你的仓库来构建和测试代码，因为它与 GitHub 紧密集成。当前，Travis CI 向开源项目提供了免费计划。很多 Python 开源项目都使用 Travis CI 来确保程序能够在多个 Python 版本和平台上正确地运行。

在 GitHub 仓库中配置 Travis CI 很容易，只需在其中包含一个.travis.yml 文件（其中包含项目构建指令），再前往 Travis CI 网站注册一个账户并激活这个仓库。

下面是一个高性能应用程序的.travis.yml 文件，这个文件包含构建并运行软件的指令，这些指令是使用 YAML 语法在几部分中定义的。

python 部分指定要使用哪个版本的 Python。install 部分指定下载并安装 conda，以便测试和安装依赖以及设置项目。这部分并非必不可少（可转而使用 pip），但对高性能应用程序来说，conda 是一个很好的包管理器，因为它包含很有用的原生包。

script 部分包含测试项目所需的代码。在这个示例中，只运行测试和基准测试程序。

```
language: python
python:
  - "2.7"
  - "3.5"
install:
# 安装 miniconda
- sudo apt-get update
- if [[ "$TRAVIS_PYTHON_VERSION" == "2.7" ]]; then
    wget https://repo.continuum.io/miniconda/
    Miniconda2-latest-Linux-x86_64.sh -O miniconda.sh;
  else
```

```
      wget https://repo.continuum.io/miniconda/
      Miniconda3-latest-Linux-x86_64.sh -O miniconda.sh;
    fi
  - bash miniconda.sh -b -p $HOME/miniconda
  - export PATH="$HOME/miniconda/bin:$PATH"
  - hash -r
  - conda config --set always_yes yes --set changeps1 no
  - conda update -q conda
  # 安装 conda 依赖
  - conda create -q -n test-environment python=
    $TRAVIS_PYTHON_VERSION numpy pandas cython pytest
  - source activate test-environment
  # 安装 pip 依赖
  - pip install pytest-benchmark
  - python setup.py install

script:
  pytest tests/
  pytest benchmarks/
```

每当有新代码被推送（push）到 GitHub 仓库（或发生其他指定的事件）时，Travis CI 都将启动一个容器、安装依赖并运行测试套件。在开源项目中使用 Travis CI 是一种极佳的做法，因为这样可不断提供有关项目状态的反馈，还可通过经过反复考验的.travis.yml 文件提供最新的安装指令。

## 9.5　小结

为软件选择优化策略是一项复杂而微妙的任务，具体选择什么策略取决于应用程序的类型、目标平台和业务需求。本章提供了一些指南，可帮助你为自己的应用程序选择合适的软件栈。

有些高性能数值计算应用程序需要安装和部署第三方包，而这些第三方包可能需要处理外部工具和原生扩展。本章介绍了如何组织 Python 项目，包括测试、基准测试程序、文档、Cython 模块和 C 扩展；另外，还介绍了持续集成服务 Travis CI，你可使用它来不断地测试托管在 GitHub 上的项目。

最后介绍了虚拟环境和 Docker 容器，你可使用它们来以隔离的方式测试应用程序、极大地简化部署工作，以及让多位开发人员能够访问同一个平台。

Python是一种通用型编程语言，其语法清晰简洁、标准库强大，还有大量的第三方库，因而近几年人气急剧上升，在很多领域都得到了广泛应用。

本书是一本Python性能提升指南，展示了如何利用Python的原生库以及丰富的第三方库来构建健壮的应用程序。书中阐释了如何利用各种剖析器来找出Python应用程序的性能瓶颈，并应用正确的算法和高效的数据结构来解决它们；介绍了如何有效地利用NumPy、Pandas和Cython高性能地执行数值计算；解释了异步编程的相关概念，以及如何利用响应式编程实现响应式应用程序；概述了并行编程的概念，并论述了如何利用TensorFlow和Theano为并行架构编写代码，以及如何通过Dask和PySpark等技术在计算机集群上执行大规模计算。

通过学习本书，你将能够实现高性能、可伸缩的Python应用程序。

- 利用NumPy和Pandas编写高效的数值计算代码
- 利用Cython和Numba实现近似本地的性能
- 利用剖析器发现Python应用程序的瓶颈
- 利用`asyncio`和RxPy编写整洁的并发代码
- 利用TensorFlow和Theano在Python中自动实现并行性
- 利用Dask和PySpark在计算机集群上运行分布式并行算法

Packt
www.packtpub.com
图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/Python
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-48877-0
定价：59.00元

# 看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks